# Closure Representations on the .NET CLR

## Experiments and Observations

Don Syme,
Microsoft Research, Cambridge

# A real problem…

- .NET language-level interop is largely solved:
  - e.g. objects, calling conventions, exceptions
  - e.g. C# can easily call VB
  - e.g. SML can easily catch C# exceptions
  - e.g. SML.NET can understand C# classes
- BUT:
  - Scheme: Untyped function values
  - Standard ML: Typed function values
  - Haskell: Typed function values, also delayed computations
  - OCaml: Typed function values
  - Funnel: Typed function values, also delayed computations
  - Mercury: Typed function values
  - …

# A real problem…

- No interop over ANY of these constructs
  - A function value from one language cannot easily be used in another language
  - There are subtle and not-so-subtle differences between them.
  - Even if there isn't, compilers all encode them in different ways.
- The question: Can we do anything about this?

# Overview

- Definitions and Aims
- A Range of Possible Standardized Encodings
- Experimental Results
- Conclusions & Ways Forward

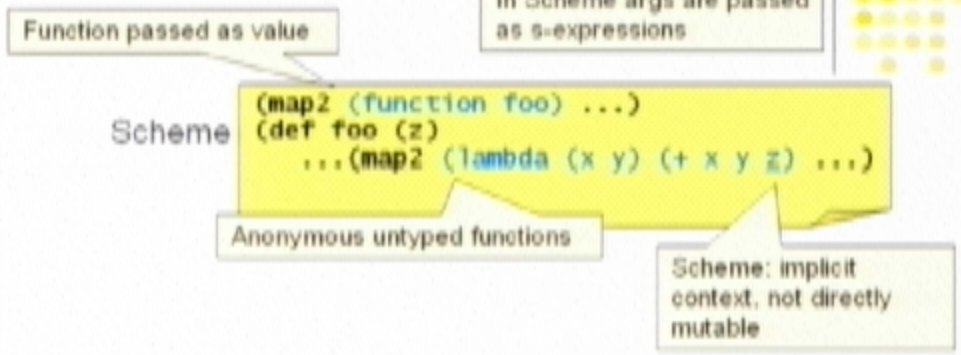
# Source Languages

# Source Languages

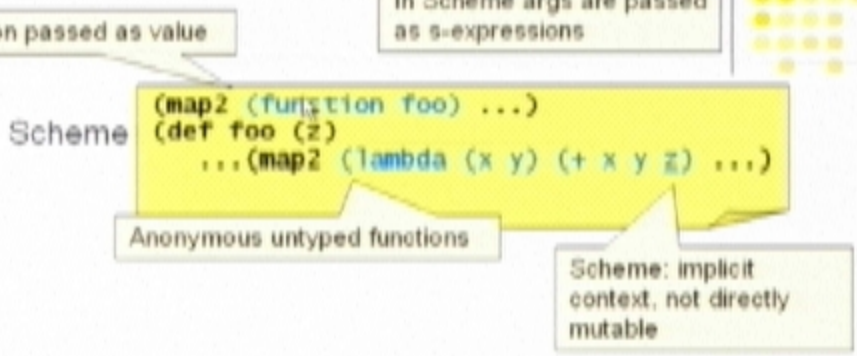
Scheme

```
(map2 (function foo) ...)
(def foo (z)
   ...(map2 (lambda (x y) (+ x y z) ...)
```

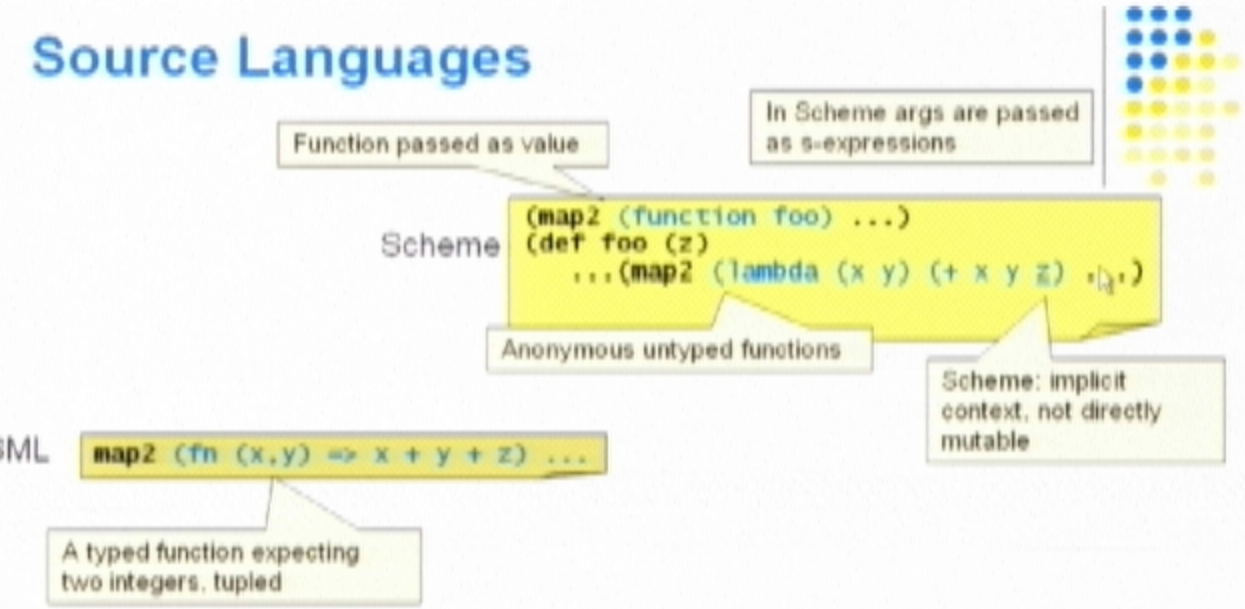
# Source Languages



# Source Languages



# Source Languages



# Source Languages

Function passed as value

In Scheme args are passed as s-expressions

Scheme

```
(map2 (function foo) ...)
(def foo (z)
   ...(map2 (lambda (x y) (+ x y z)) ...)
```

Anonymous untyped functions

Scheme: implicit context, not directly mutable

ML

```
map2 (fn (x,y) => x + y + z) ...
```

A typed function expecting two integers, tupled

Again typed

OCaml

```
map2 (fun x y -> x + y + z) ...
=== map2 (fun x -> (fun y -> x + y + z)) ...
```

Functions return functions ("currying") are very common

# Source Languages

In Scheme args are passed as s-expressions

Function passed as value

Scheme

```
(map2 (function foo) ...)
(def foo (z)
   ...(map2 (lambda (x y) (+ x y z)) ...)
```

Anonymous untyped functions

Scheme: implicit context, not directly mutable

ML

```
map2 (fn (x,y) => x + y + z) ...
```

A typed function expecting two integers, tupled

Again typed

OCaml

```
map2 (fun x y -> x + y + z) ...
=== map2 (fun x -> (fun y -> x + y + z)) ...
```

Functions return functions ("currying") are very common

Haskell

```
map2 (\x y. f x + f y + f z) ...
```

Again typed, again lots of currying, also lots of delayed computations

# Definitions

- Closures: "functions in a context"
  - Context = Environment = Free Variables
- Closure expressions: declarations that specify new inner functions and their contexts (context is usually implicit)
- Function value: "code pointer + data" = an object allocated to hold pointer(s) to the function code plus the environment
- Function application: invoking a function value
- Function type: a general class of types that function values belong to.

# Aims

- Fully-specified encoding of closures into MS-IL
- Easy to generate
- Supports separate compilation (no global analysis needed).
- Fast enough for, e.g. OCaml
- Fits with MS-IL polymorphism/generics

# Example

GC#

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (fun y -> rename (valOld,valNew,y));

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v;
```

Functional language

# Example

GC#

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

A function object is passed in.

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (fun y -> rename (valOld,valNew,y));

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v;
```

Functional language

# Example

GC#

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

A function object is passed in.

Here the same function object is invoked many times.

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (fun y -> rename (valOld,valNew,y));

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v;
```

Functional language

# Example

GC#

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

A function object is passed in.

Here the same function object is invoked many times.

Here we pass in a function. Each time we call "table.map" a new closure will be allocated.

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (fun y -> rename (valOld,valNew,y));

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v;
```

Functional language

# Example

GC#

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

A function object is passed in.

Here the same function object is invoked many times.

Here we pass in a function. Each time we call "table.map" a new closure will be allocated.

Functional language

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (fun y -> rename (valOld,valNew,y));

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v;
```

This closure expression uses elements from the environment. Each time we call "map" a new function object will be allocated.

# Example

GC#

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

A function object is passed in.

Here the same function object is invoked many times.

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (fun y -> rename (valOld,valNew,y));

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v;
```

Here we pass in a function. Each time we call "table.map" a new closure will be allocated.

Functional language

This closure expression uses elements from the environment. Each time we call "map" a new function object will be allocated.

When invoked the function object calls this function. It can have side-effects.

# Example

GC#

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (new clo[valOld,valNew]);

let clo[valOld,valNew](y) = rename (valOld,valNew,y);

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

Functional language

# Example

GC#

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

A function object is passed in.

This is an encoded generic function type

Here the same function object is invoked many times.

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (fun y -> rename (valOld,valNew,y));

let rename(valOld,valNew,v) =
   if (v = valOld) then
      rename++;
      valNew;
   else v;
```

Here we pass in a function. Each time we call "table.map" a new closure will be allocated.

Functional language

This closure expression uses elements from the environment. Each time we call "map" a new function object will be allocated.

When invoked the function object calls this function. It can have side-effects.

# Example

GC#

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (new clo[valOld,valNew]);

let clo[valOld,valNew](y) = rename (valOld,valNew,y);

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

Functional language

# Example

GC#

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (new clo[valOld,valNew]);

let clo[valOld,valNew](y) = rename (valOld,valNew,y);

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

Functional language

Assume all closure expressions are named with the environment annotated.

# Part 2: Implementation Choices

# Example

GC#

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (new clo[valOld,valNew]);

let clo[valOld,valNew](y) = rename (valOld,valNew,y);

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

Functional language

Assume all closure expressions are named with the environment annotated.

# Part 2: Implementation Choices

# Overview

- Note: for typed functional languages we really assume Generics have been added to the CLR

Generics == Parametric Polymorphism == Templates

# Choices & Non-choices

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (new clo[valOld,valNew]);

let clo[valOld,valNew](y) = rename (valOld,valNew,y);

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

# Choices & Non-choices

Non-choice: Function types must be structural

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (new clo[valOld,valNew]);

let clo[valOld,valNew](y) = rename (valOld,valNew,y);

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

# Choices & Non-choices

Non-choice: Function types must be structural

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

Encoding function types as generic MS-IL types seems obvious: structural equivalence comes for free.

Note: could also build these into MS-IL. e.g. make delegates structural.

Note: System.Func could still be a class, value class, interface or a general class of delegates

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (new clo[valOld,valNew]);

let clo[valOld,valNew](y) = rename (valOld,valNew,y);

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

# Choices & Non-choices

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

Non-choice: functions objects are heap-allocated.

```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (new clo[valOld,valNew]);

let clo[valOld,valNew](y) = rename (valOld,valNew,y);

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

# Choices & Non-choices

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

Non-choice: functions objects are heap-allocated.


```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (new clo[valOld,valNew]);

let clo[valOld,valNew](y) = rename (valOld,valNew,y);

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

# Choices & Non-choices

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

Non-choice: functions objects are heap-allocated.


```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (new clo[valOld,valNew]);

let clo[valOld,valNew](y) = rename (valOld,valNew,y)

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

Choice: How is the environment stored w.r.t . the function object?

The relevant parts of the environment must be somehow accessed via fields of the function object

# Choices & Non-choices

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

Non-choice: fu
heap-allocated

Choice: How do we get from the function value to the code? i.e. how is the code is "stored" in/accessed from function object.

Getting from A to B is an indirect call. On the .NET CLR, class vtables, interface vtables, delegates or code pointers are the only choices.


```
let table = new Table<int,string>(10);
let rename = 0;

let process_update (valOld,valNew) =
   table.map (new clo[valOld,valNew]);

let clo[valOld,valNew](y) = rename (valOld,valNew,y)

let rename(valOld,valNew,v) =
   if (v = valOld)
      rename++;
      valNew
   else v
```

Choice: How is the environment stored w.r.t . the function object?

The relevant parts of the environment must be somehow accessed via fields of the function object

# The Simplest Implementation

- Encoding into virtual methods

```
abstract class System.Func<A,B> {
    virtual B apply(A);
}

class clo : System.Func<string,string> {
   private string valOld;
   private string valNew;
   override string apply(string) { ... }
}

   .. new clo(...) ...
   .. f.apply() ...
```

- Code = virtual method
- Environment: stored inline in object

# The Simplest Implementation

- Encoding into virtual methods

This is the space of function types.

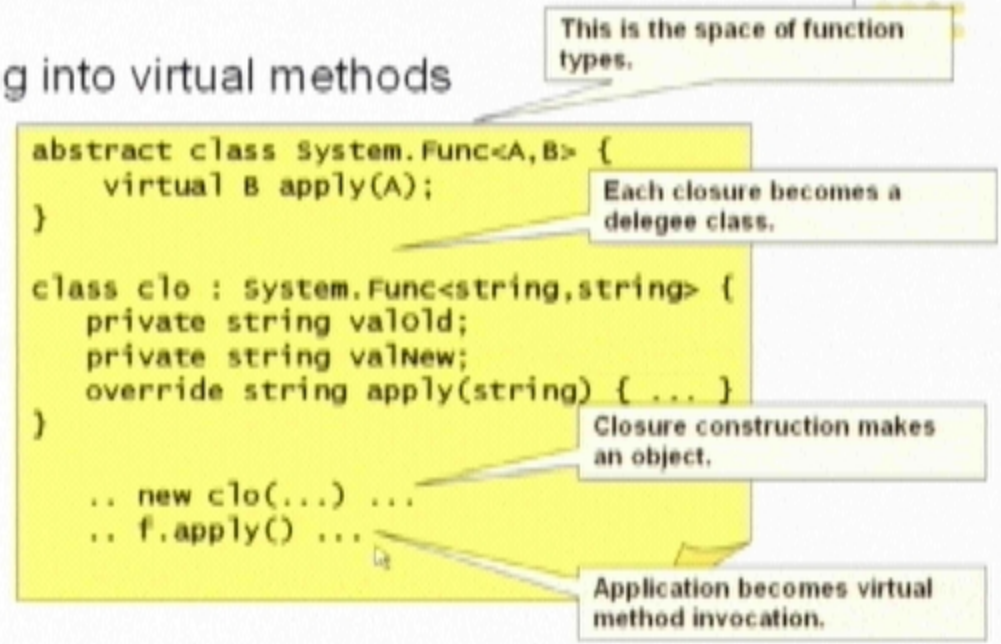
```
abstract class System.Func<A,B> {
    virtual B apply(A);
}

class clo : System.Func<string,string> {
   private string valOld;
   private string valNew;
   override string apply(string) { ... }
}

   .. new clo(...) ...
   .. f.apply() ...
```

Each closure becomes a delegee class.

- Code = virtual method
- Environment: stored inline in object

# The Simplest Implementation

- Encoding into virtual methods




- Code = virtual method
- Environment: stored inline in object

# The Delegate Implementation

- Encoding into virtual methods

```
delegate B System.Func<A,B>(A);

class clo {
   private string valold;
   private string valNew;
   string apply(string) { ... }
}


   ... new clo(...)
   ... new System.Func<string,string>(clo.apply)

   ... f.Invoke("abc");
```

- Code = method on delegee
- Environment: stored in delegee

# The Delegate Implementation

- Encoding into virtual methods

```
delegate B System.Func<A,B>(A);

class clo {
   private string valOld;
   private string valNew;
   string apply(string) { ... }
}


   ... new clo(...)
   ... new System.Func<string,string>(clo.apply)

   ... f.Invoke("abc");
```

This is the space of function types.

Each closure becomes a delegee class.

- Code = method on delegee
- Environment: stored in delegee

# The Delegate Implementation

- Encoding into virtual methods

```
delegate B System.Func<A,B>(A);

class clo {
   private string valOld;
   private string valNew;
   string apply(string) { ... }
}


   ... new clo(...)
   ... new System.Func<string,string>(clo.apply)

   ... f.Invoke("abc");
```

This is the space of function types.

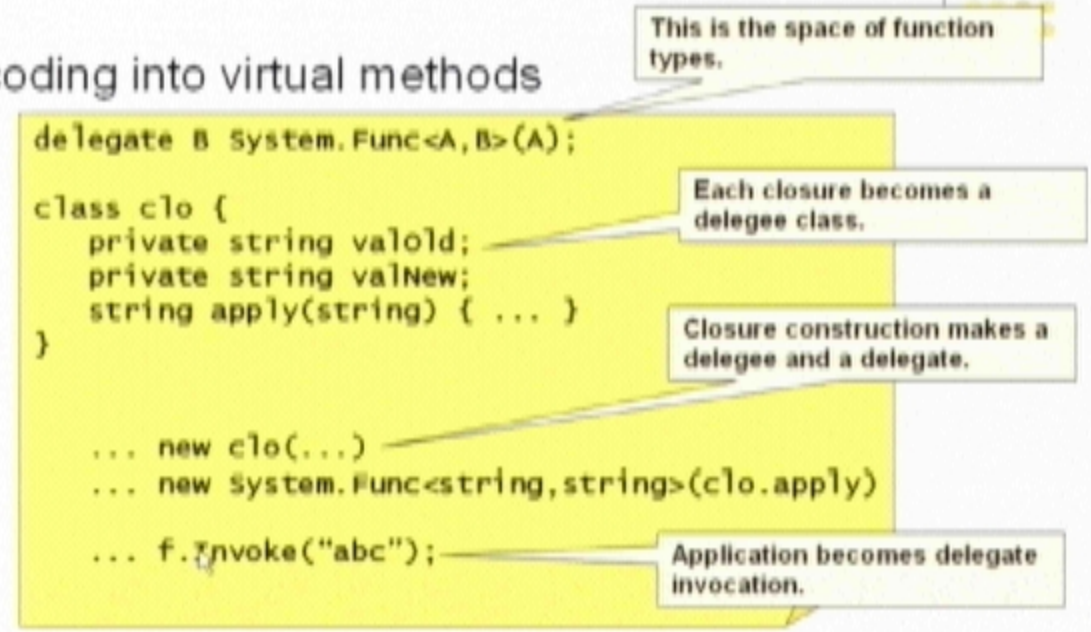
Each closure becomes a delegee class.

Closure construction makes a delegee and a delegate.

- Code = method on delegee
- Environment: stored in delegee

# The Delegate Implementation

- Encoding into virtual methods




- Code = method on delegee
- Environment: stored in delegee

# A Library Implementation

- Encode into:
  - MS-IL C-style function pointers;
  - Closure "template" classes

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ ... }

class System.Clo<E,A,B> : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code);
   public E;
}
```

# A Library Implementation

- Encode into:
  - MS-IL C-style function pointers;
  - Closure "template" classes

This is the class of function types.

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ ... }

class System.Clo<E,A,B> : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code);
   public E;
}
```

# A Library Implementation

- Encode into:
  - MS-IL C-style function pointers;
  - Closure "template" classes

This is the class of function types.

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ ... }

class System.Clo<E,A,B> : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code);
   public E;
}
```

This is a helper method for application.

# A Library Implementation

- Encode into:
  - MS-IL C-style function pointers;
  - Closure "template" classes

This is the class of function types.

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ ... }

class System.Clo<E,A,B> : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code);
   public E;
}
```

This is a helper method for application.

This class is a template for all closures.

# A Library Implementation

- Encode into:
  - MS-IL C-style function pointers;
  - Closure "template" classes

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ ... }

class System.Clo<E,A,B> : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code);
   public E;
}
```

This is the class of function types.

This is a helper method for application.

This class is a template for all closures.

This is a "C-function pointer type" in MS-IL

# A Library Implementation

- Encode into:
  - MS-IL C-style function pointers;
  - Closure "template" classes

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ ... }

class System.Clo<E,A,B> : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code);
   public E;
}
```

This is the class of function types.

This is a helper method for application.

This class is a template for all closures.

This is the environment.

This is a "C-function pointer type" in MS-IL

# A Library Implementation

- Encode into:
  - MS-IL C-style function pointers;
  - Closure "template" classes

```
abstract class System.Func<A,B> {
   void * code;
}

static method B app<A,B>(System.Func<A,B>, A x)
{ ... }

class System.Clo<E,A,B> : System.Func<A,B> {
   Clo(E env, method B *(E env,A x) code);
   public E;
}
```

This is the class of function types.

This is a helper method for application.

This class is a template for all closures.

This is the environment.

This is a "C-function pointer type" in MS-IL

Client code is verifiable, but the implementation of this module can use unsafe tricks if needed. Presume the library is code-signed.

# Part 3: Further Details of Implementation Choices

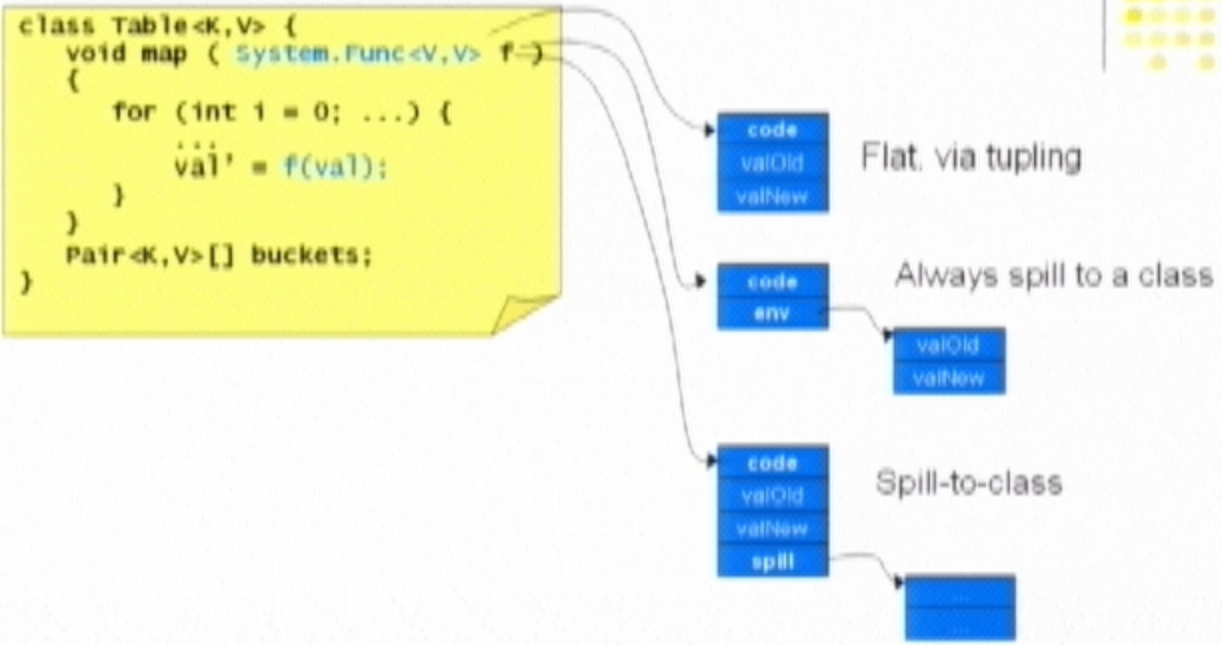
# Choices: Environments

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```


Flat, via tupling

Always split to a class?

# Choices: Environments



# Choices: Environments

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```

For closure templates, a flat effect can be achieved by using "tupling", e.g. instantiating "E" with, e.g. Pair<int,int> or Tuple4<int,int,int,int>


Flat, via tupling


Always spill to a class

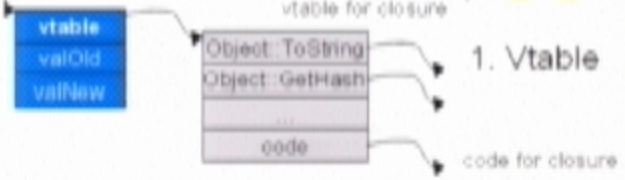


Spill-to-class

In our tests this is a new class per closure when needed

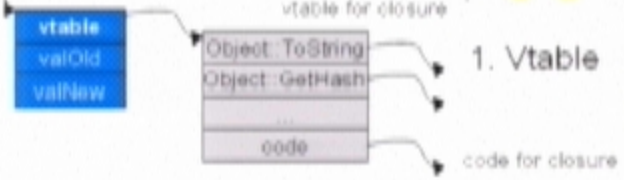
# Choices: Code

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```



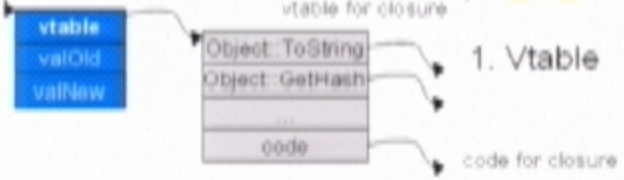
# Choices: Code

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```



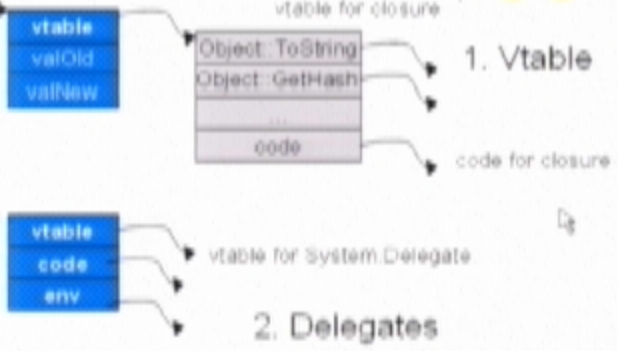
# Choices: Code

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```



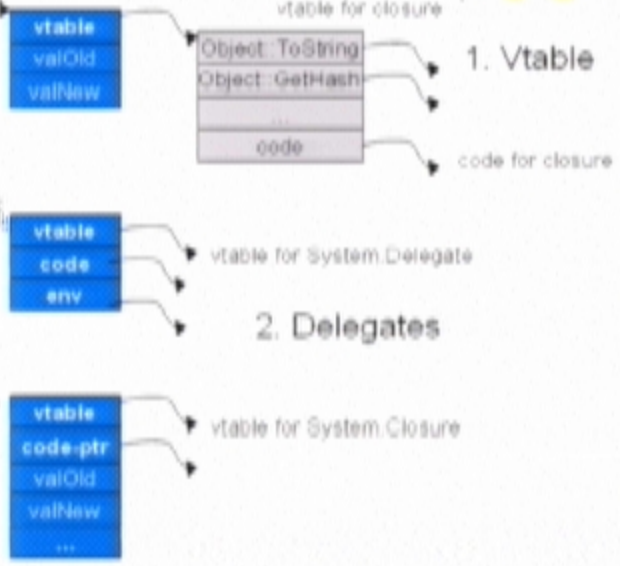
# Choices: Code

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```



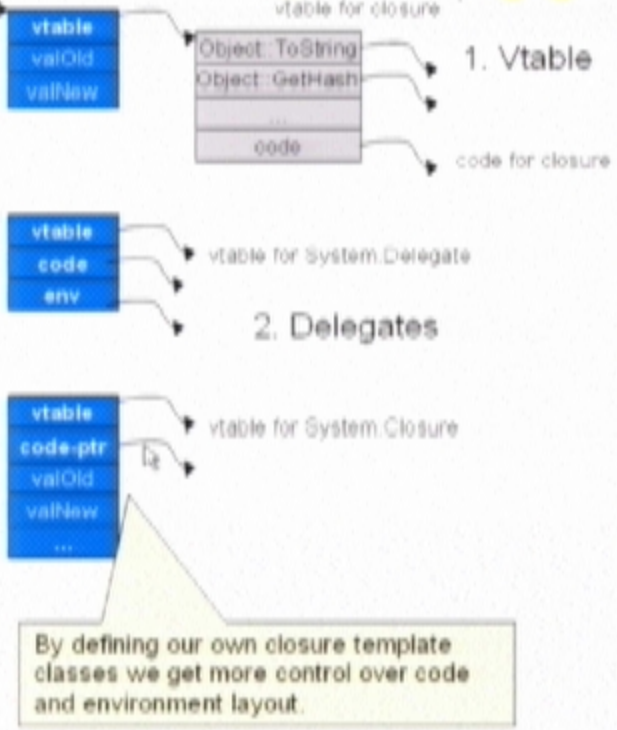
# Choices: Code

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```



# Choices: Code

```
class Table<K,V> {
   void map ( System.Func<V,V> f )
   {
      for (int i = 0; ...) {
         ...
         val' = f(val);
      }
   }
   Pair<K,V>[] buckets;
}
```



# Choices: Multiple Entries

Scheme

```
(map2 (lambda (x y) (+ x y z)) ...

(def map2 (f l1 l2) ...
      (f (car l1) (car l2)) ...
```

```
(def add2 (f x l2) ...
      (f x 4) ...
```

# Choices: Multiple Entries

Optimized functional language implementations play multiple entry point games.

Scheme

```
(map2 (lambda (x y) (+ x y z)) ...

(def map2 (f l1 l2) ...
      (f (car l1) (car l2)) ...
```

This call passes 2 objects to a function expecting 2 objects. No need to allocate arguments as a S-expr!

Solution: extra entry point on closures for passing 2 objects

This call will then go much faster.

```
(def add2 (f x l2) ...
      (f x 4) ...
```

# Choices: Multiple Entries

Optimized functional language implementations play multiple entry point games.

Scheme

```
(map2 (lambda (x y) (+ x y z)) ...

(def map2 (f l1 l2) ...
      (f (car l1) (car l2)) ...
```

This call passes 2 objects to a function expecting 2 objects. No need to allocate arguments as a S-expr!

Solution: extra entry point on closures for passing 2 objects

This call will then go much faster.

```
(def add2 (f x l2) ...
      (f x 4) ...
```

Also other entry point, e.g. 2 integers

# Choices: Multiple Entries

Optimized functional language implementations play multiple entry point games.

Scheme

```
(map2 (lambda (x y) (+ x y z)) ...

(def map2 (f l1 l2) ...
      (f (car l1) (car l2)) ...
```

This call passes 2 objects to a function expecting 2 objects. No need to allocate arguments as a S-expr!

Solution: extra entry point on closures for passing 2 objects

This call will then go much faster.

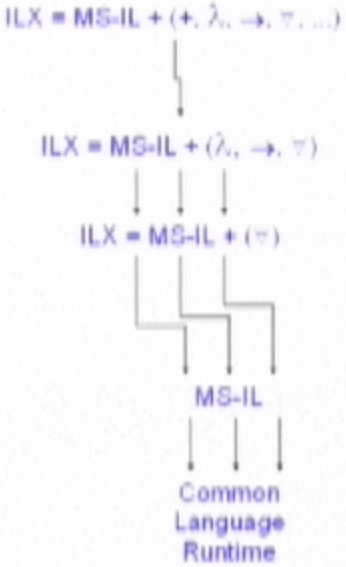
```
(def add2 (f x l2) ...
      (f x 4) ...
```

Also other entry point, e.g. 2 integers

# Part 4: Experimental Results

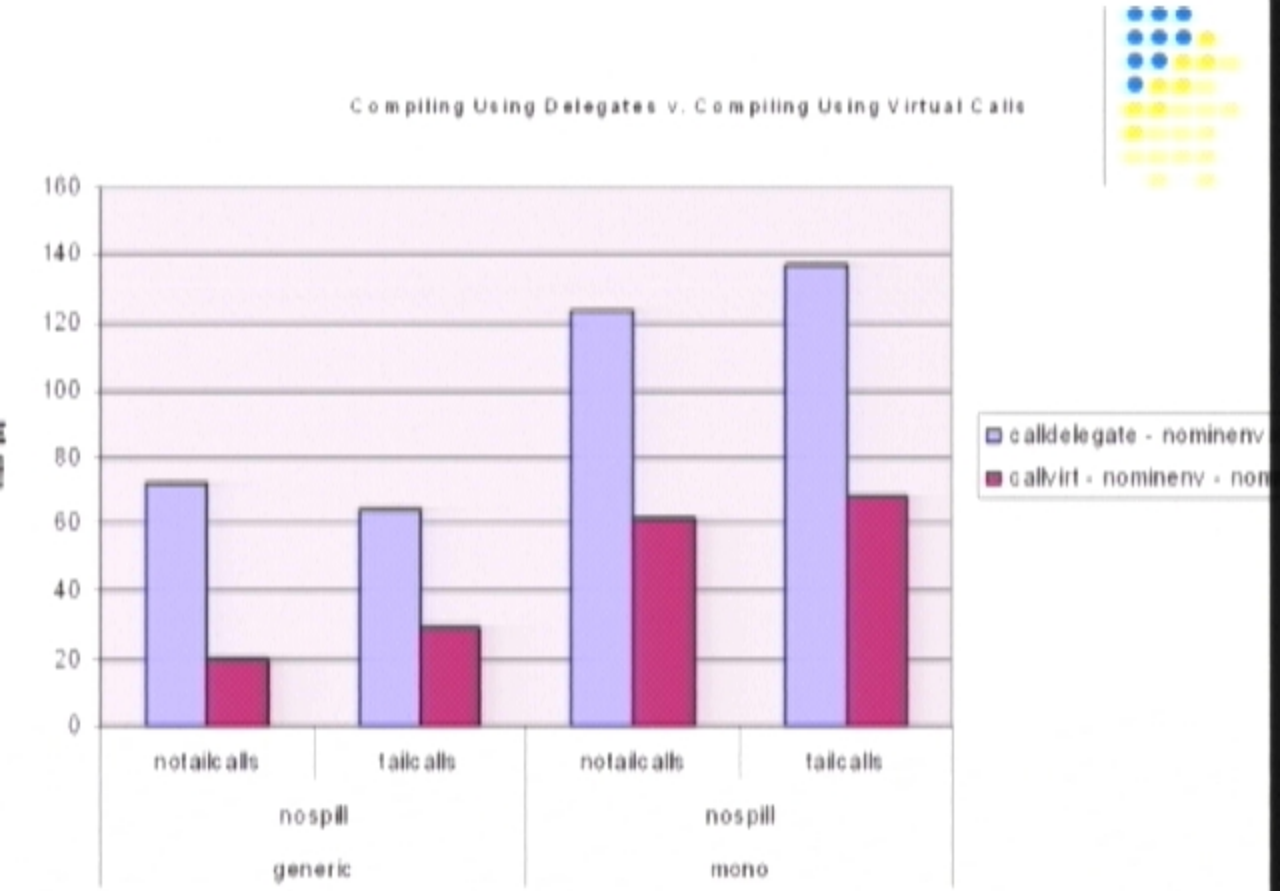
# ILX Implementations by Translation

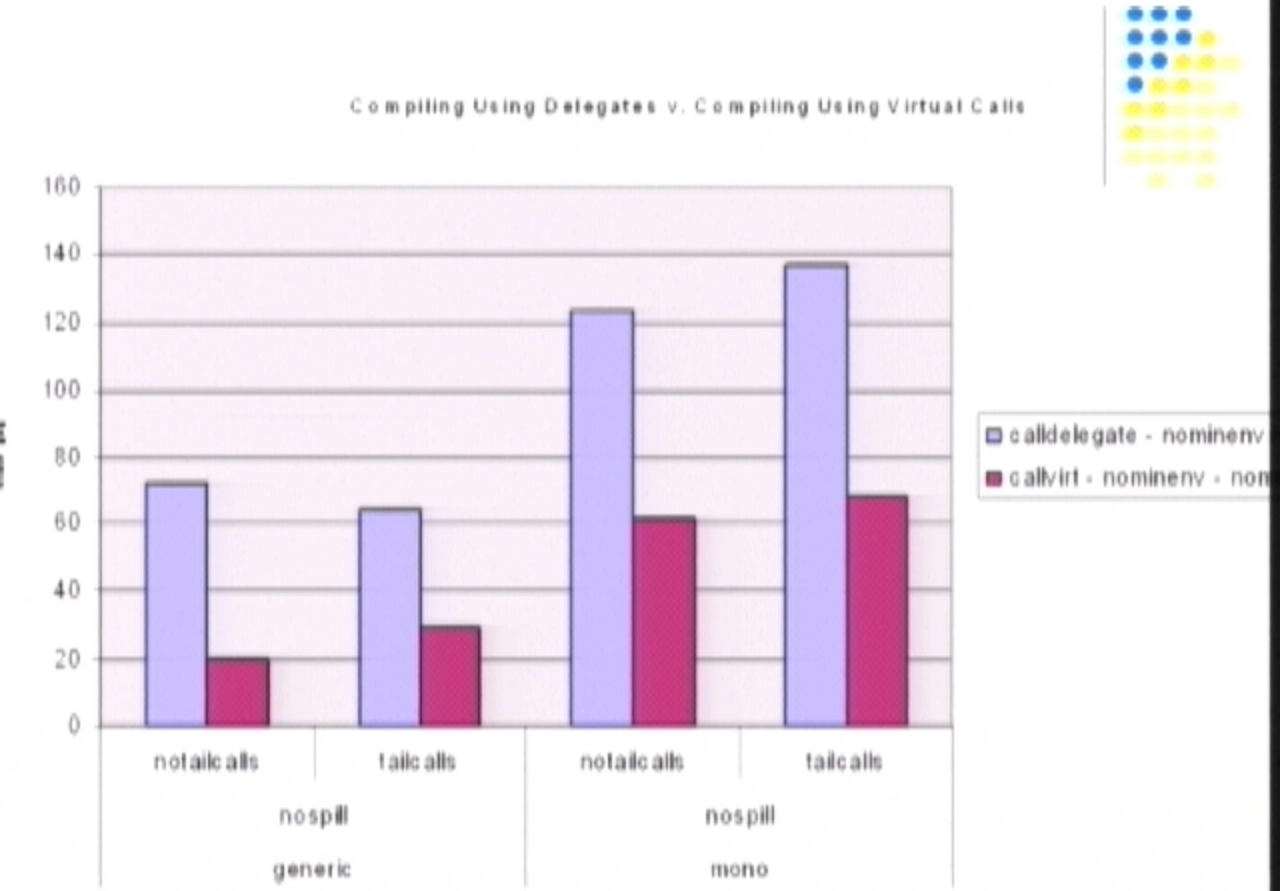
Haskell SML

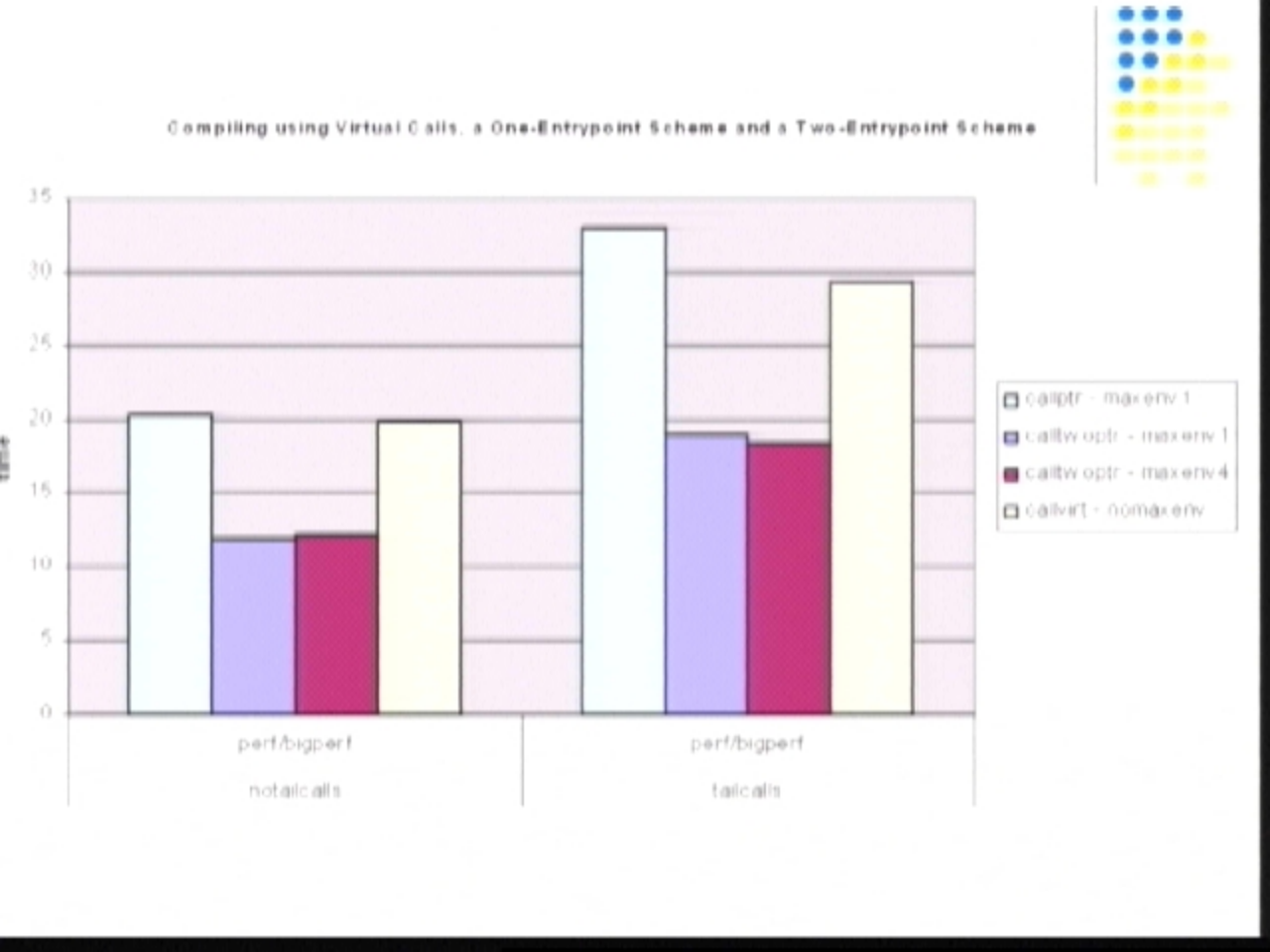


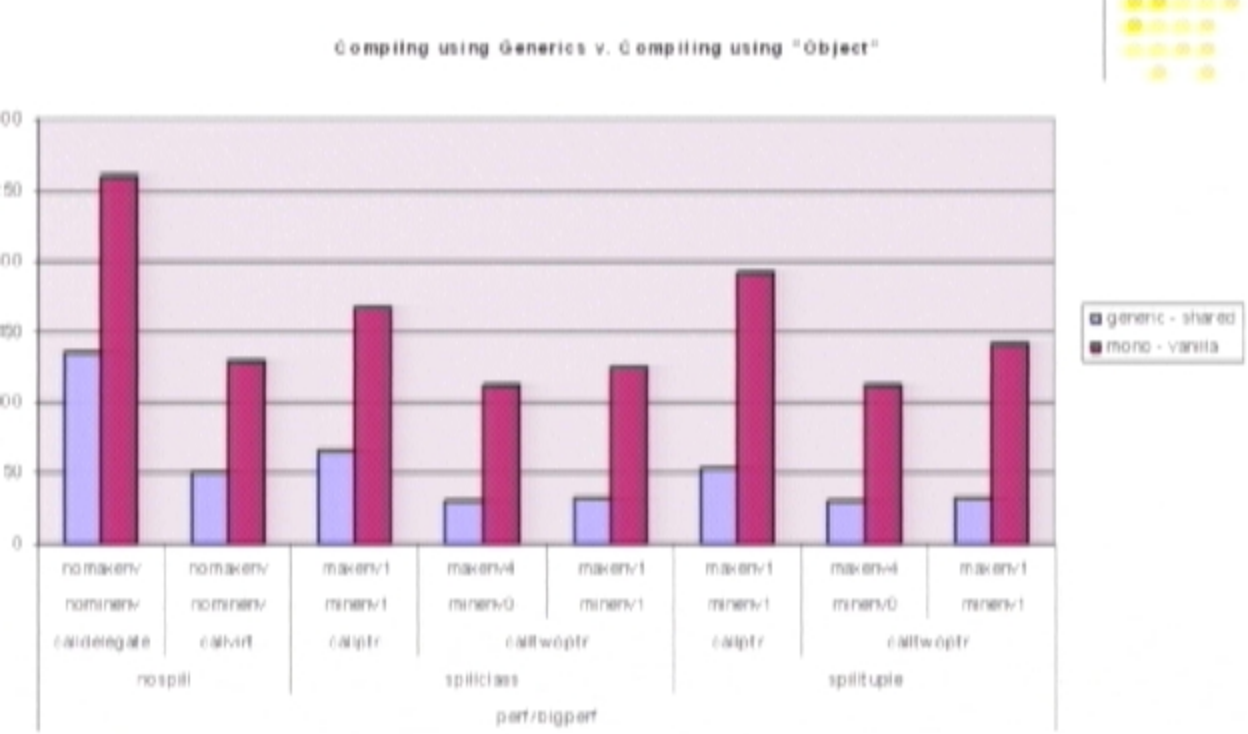
# Simulated Performance Load

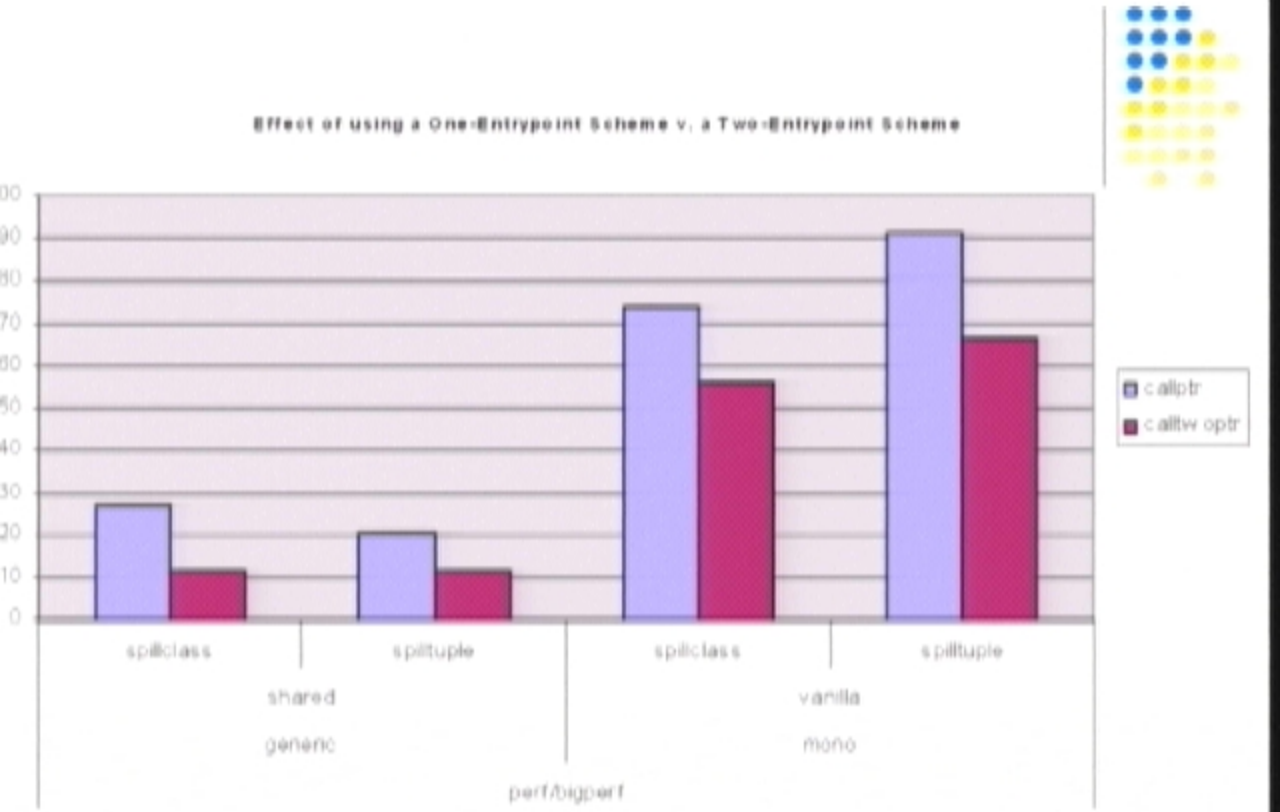
- Designed to be (very roughly) like the indirect calling load of a typical OCaml program:
  - 80% of closures take 1 argument
  - 15% take 2
  - rest take 3
  - 60% of closures have 0 free variables
  - 30% of closures have 1 free variable
  - 10% of closures have 2 free variables
  - All calls are indirect
  - Each closure used to create a varying number of function values
  - Each function value called a varying number of times
- BEST TIME =~ 11s

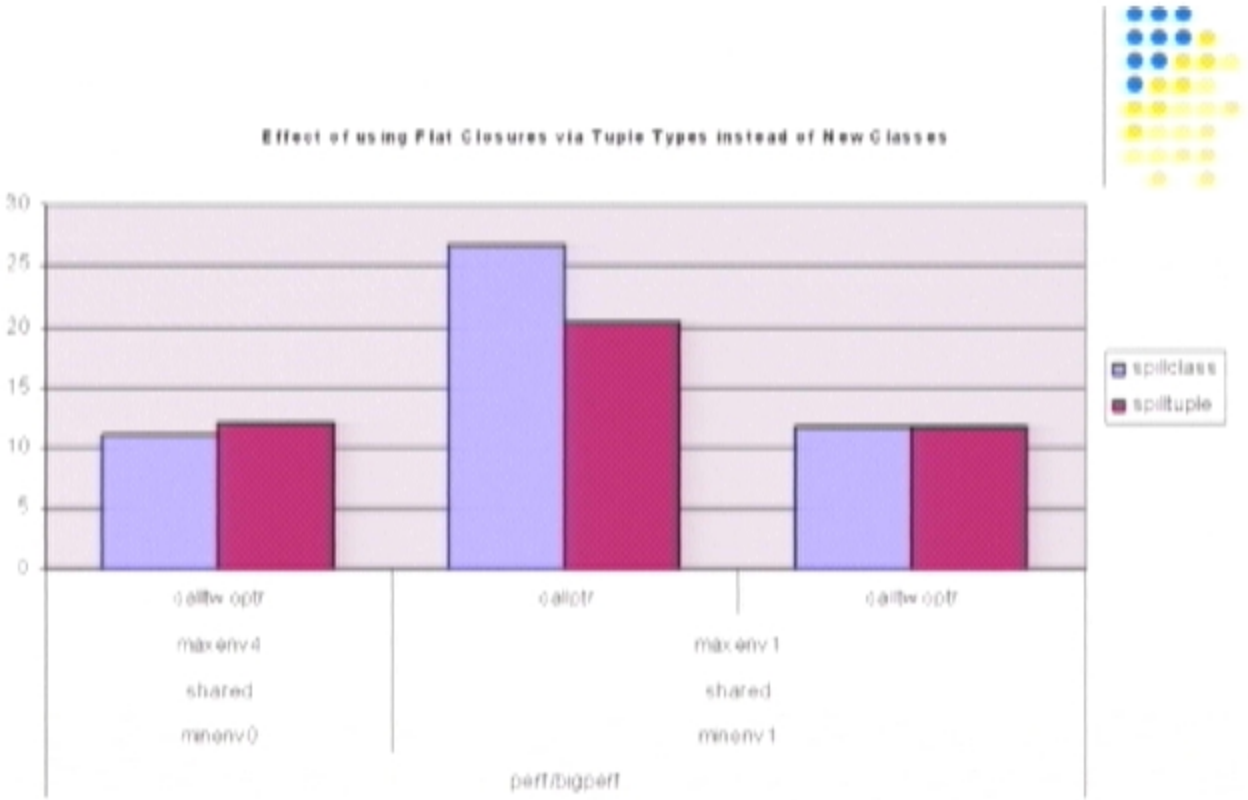

Compiling Using Delegates v. Compiling Using Virtual Calls
160
140
120
100
80
60
40
20
0
notailcalls
tailcalls
notailcalls
tailcalls
nospill
generic
nospill
mono
calldelegate - nominenv
callvirt - nominenv - non

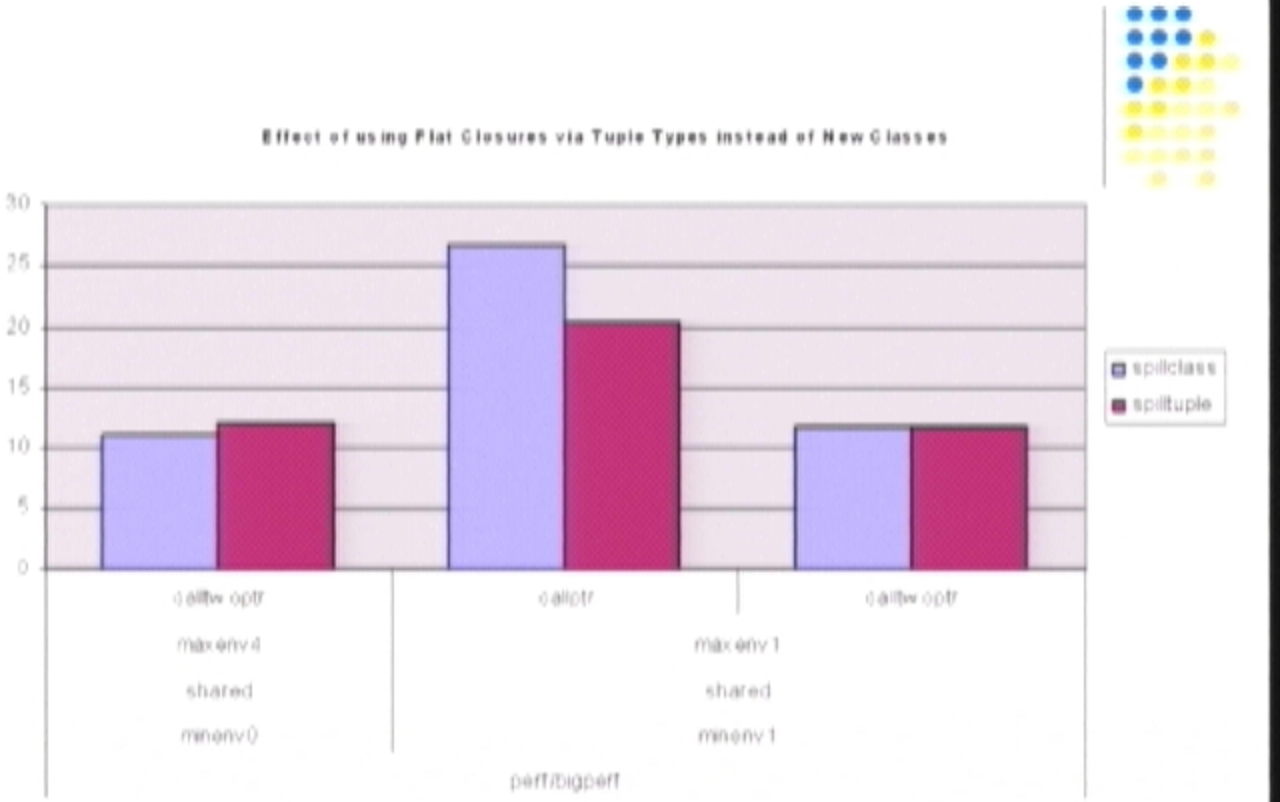

Compiling Using Delegates v. Compiling Using Virtual Calls
160
140
120
100
80
60
40
20
0
notailcalls
tailcalls
notailcalls
tailcalls
nospill
generic
nospill
mono
calldelegate - nominenv
callvirt - nominenv - non

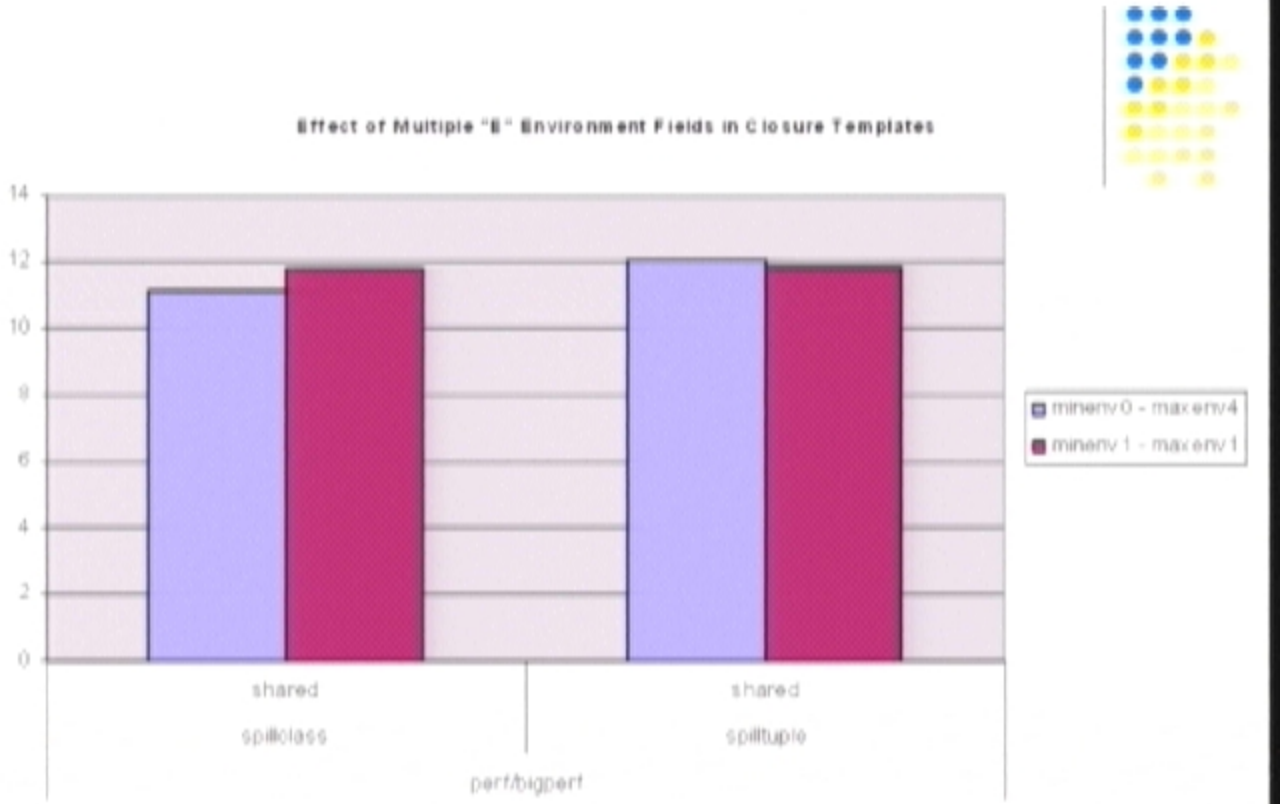

Compiling using Virtual Calls, a One-Entrypoint Scheme and a Two-Entrypoint Scheme
35
30
25
20
15
10
5
0
callptr - maxenv 1
callfw optr - maxenv 1
callfw optr - maxenv 4
callvirt - nomaxenv
perf/bigperf
notailcalls
perf/bigperf
tailcalls

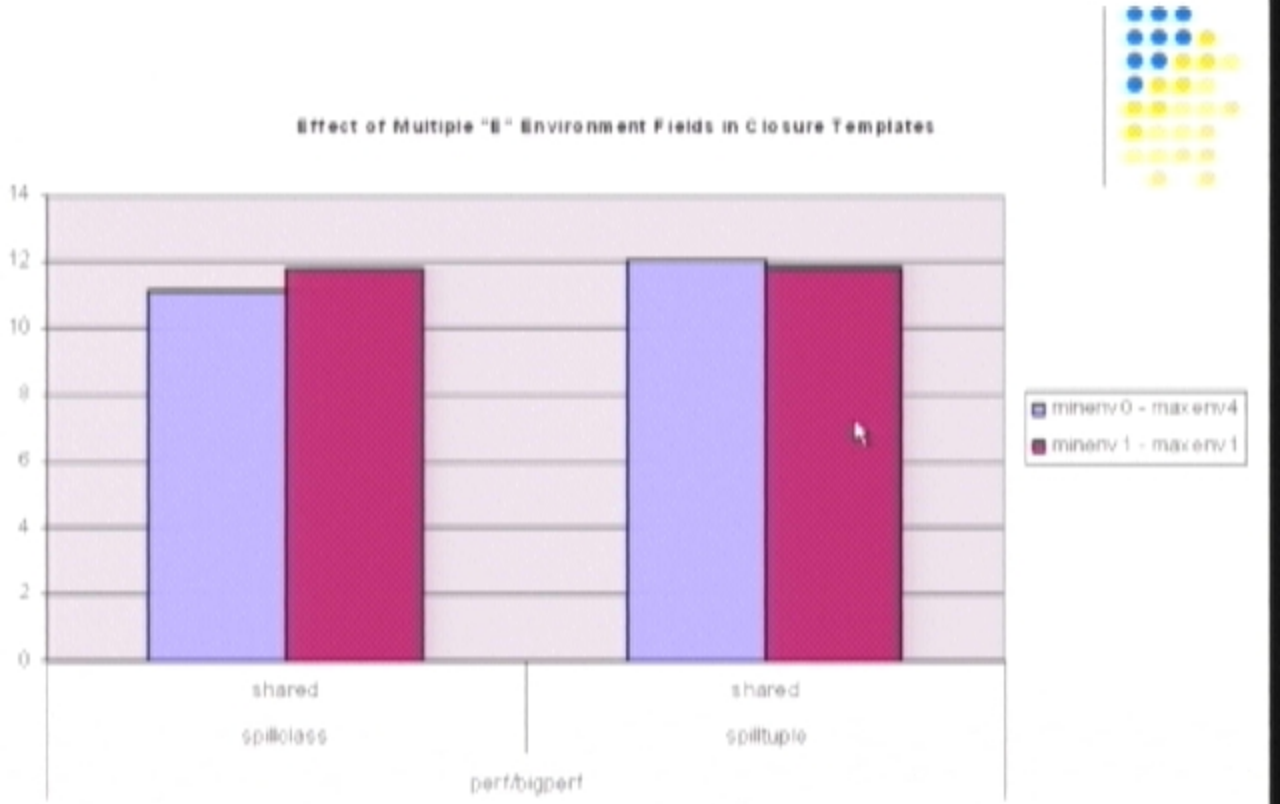

Compiling using Generics v. Compiling using "Object"
00
50
00
50
00
50
0
generic - shared
mono - vanilla
nomakenv
nomnenv
/aldelegate
nomakenv
nomnenv
/allvirt
makenv/1
mnenv/1
/allptr
makenv/4
mnenv/0
makenv/1
mnenv/1
/alltwoptr
makenv/1
mnenv/1
/allptr
makenv/4
mnenv/0
makenv/1
mnenv/1
/alltwoptr
nospli
splitclass
splitupie
perf/bigperf

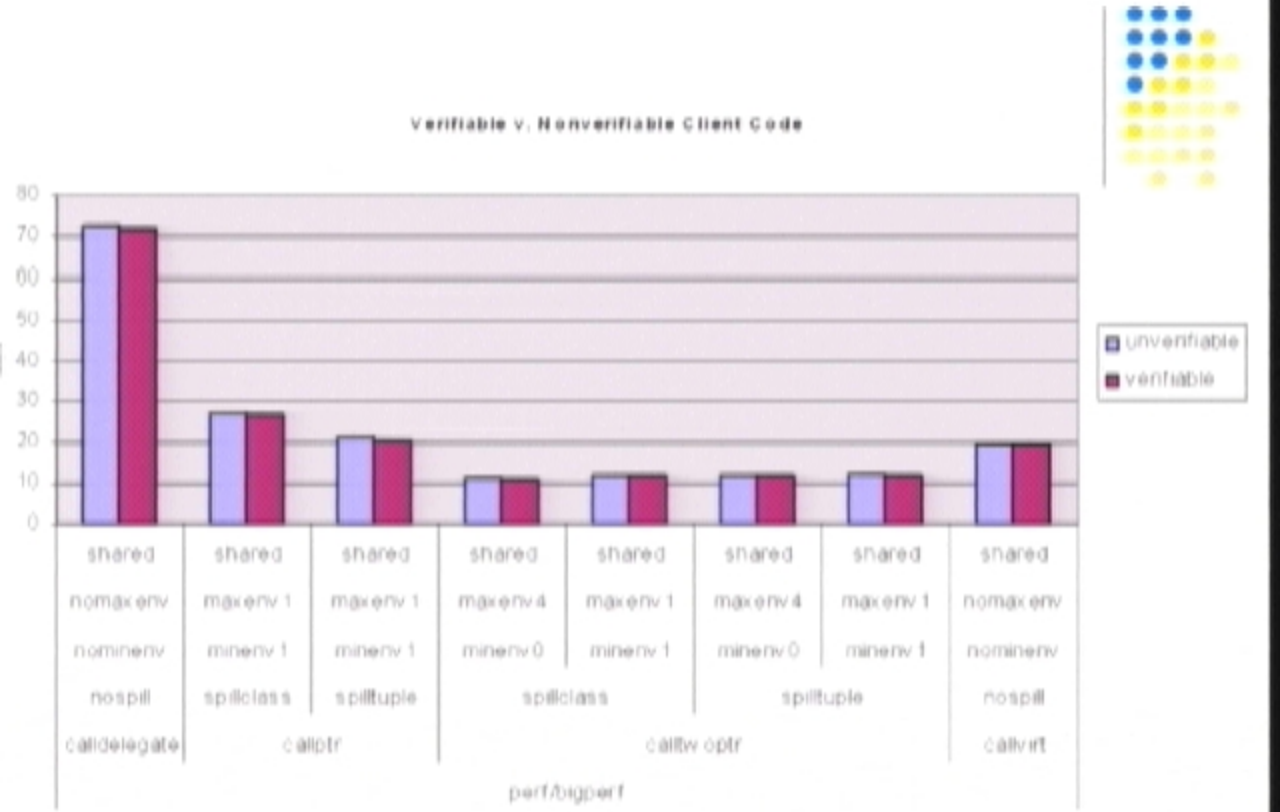

Effect of using a One-Entrypoint Scheme v. a Two-Entrypoint Scheme
callptr
callvw optr
spliclass
spiltuple
spliclass
spiltuple
shared
generic
vanilla
mono
perf/bigperf


Effect of using Flat Closures via Tuple Types instead of New Classes
30
25
20
15
10
5
0
splclass
spltuple
callfw optr
callctr
callfw optr
max env 4
shared
min env 0
max env 1
shared
min env 1
perf/bigperf


Effect of using Flat Closures via Tuple Types instead of New Classes
30
25
20
15
10
5
0
spillclass
spilltuple
callfw optr
callctr
callfw optr
max env 4
max env 1
shared
shared
min env 0
min env 1
perf/bigperf


Effect of Multiple "E" Environment Fields in Closure Templates
14
12
10
8
6
4
2
0
shared
spillclass
shared
spilltuple
perf/bigperf
minenv 0 - maxenv 4
minenv 1 - maxenv 1


Effect of Multiple "E" Environment Fields in Closure Templates
14
12
10
8
6
4
2
0
shared
spillclass
shared
spilltuple
perf/bigperf
minenv 0 - maxenv 4
minenv 1 - maxenv 1


Verifiable v. Nonverifiable Client Code
80
70
60
50
40
30
20
10
0
unverifiable
verifiable
shared
nomaxenv
nominenv
nospill
callidelegate
shared
maxenv 1
minenv 1
spillclass
shared
maxenv 1
minenv 1
spilltuple
callptr
shared
maxenv 4
minenv 0
shared
maxenv 1
minenv 1
spillclass
shared
maxenv 4
minenv 0
shared
maxenv 1
minenv 1
spilltuple
callfw optr
shared
nomaxenv
nominenv
nospill
callvirt
perf/bigperf


Verifiable v. Nonverifiable Client Code
80
70
60
50
40
30
20
10
0
unverifiable
verifiable
shared
nomaxenv
nominenv
nospill
calldelegate
shared
maxenv 1
minenv 1
spillclass
shared
maxenv 1
minenv 1
spilltuple
callptr
shared
maxenv 4
minenv 0
shared
maxenv 1
minenv 1
spillclass
shared
maxenv 4
minenv 0
shared
maxenv 1
minenv 1
spilltuple
callvw optr
shared
nomaxenv
nominenv
nospill
callvirt
perf/bigperf

# Conclusions

1. V1 Delegates suck
2. V1 Tailcalls suck
3. Generics are an extremely useful compilation device
4. Multiple entry points are feasible and worth it
5. Flat closures are feasible and worth it
6. Multiple environment slots are not worth it

7. A library-based approach is feasible

# Conclusions

1. V1 Delegates suck
2. V1 Tailcalls suck
3. Generics are an extremely useful compilation device
4. Multiple entry points are feasible and worth it
5. Flat closures are feasible and worth it
6. Multiple environment slots are not worth it

7. A library-based approach is feasible

# Related Concepts

- Classes/Objects
  - Context is explicit (the fields)
  - Contain much more: multiple methods, static methods, reflection, ...
- Java Inner Class: "class in a context"
  - Context is explicit
  - Class can have many methods
- C# Delegates
  - Context is explicit
  - Code and data are tied
  - Also asynchronous invocation etc.
- Method pointer
  - Object + pointer to method
  - = Unboxed Delegate
  - Full closures are much more useful

# Looking Forward

Ways ahead:

1. Thoroughly investigate performance for real programs, plus a range of languages

2. V2 delegates =

V1 delegates
+
Basic perf improvements
+
Delegate to static, external methods
+
Environments inline in delegate
+
Multiple entry points

# Summary

Problems:

1. Reusing our generics design gives exact runtime type semantics.

2. The most efficient standardized encoding/shared library of closures is quite complex.

Future research:

1. Thoroughly investigate performance for closures

2. Perhaps propose standard encodings for other constructs

# Questions?

# Runtime Verification and .NET

Michael Barnett
Wolfram Schulte
Foundations of Software Engineering

# Testing and Verification?

"Computer Science is akin to astronomy and astrology, but it lacks the precision of the former and the popularity of the latter."

# Specifications are Good…

# Specifications are Good…

- So why doesn't anyone write them?

# Specifications are Good…

- So why doesn't anyone write them?
  - Limited usefulness: dead documents

# Specifications are Good…

- So why doesn't anyone write them?
  - Limited usefulness: dead documents
- But some people **do** write them

# Specifications are Good…

- So why doesn't anyone write them?
  - Limited usefulness: dead documents
- But some people **do** write them
  - Programmers write assertions

# Specifications are Good…

- So why doesn't anyone write them?
  - Limited usefulness: dead documents
- But some people **do** write them
  - Programmers write assertions
  - Programmers write error checking

# Specifications are Good…

- So why doesn't anyone write them?
  - Limited usefulness: dead documents
- But some people **do** write them
  - Programmers write assertions
  - Programmers write error checking
  - Programmers write exception handling

# Specifications are Good…

- So why doesn't anyone write them?
  - Limited usefulness: dead documents
- But some people **do** write them
  - Programmers write assertions
  - Programmers write error checking
  - Programmers write exception handling
  - Programmers write code!

# It's the level, stupid…

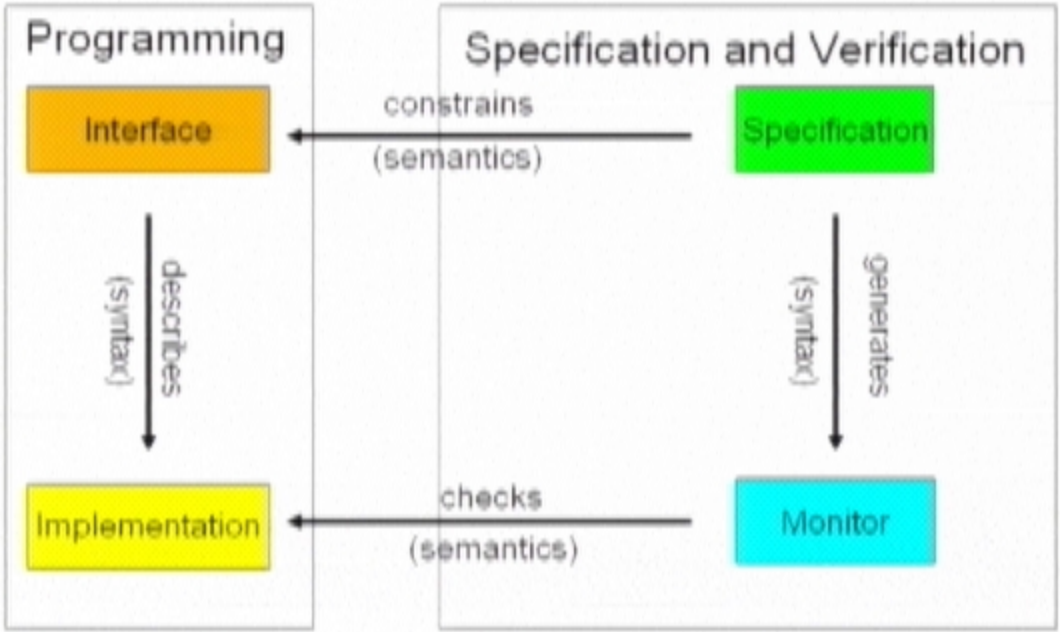
- Specifications would be more useful if they were:
  - At a higher level
  - Available separate from source code
  - Enforceable

# It's the level, stupid…

- Specifications would be more useful if they were:
  - At a higher level
  - Available separate from source code
  - Enforceable

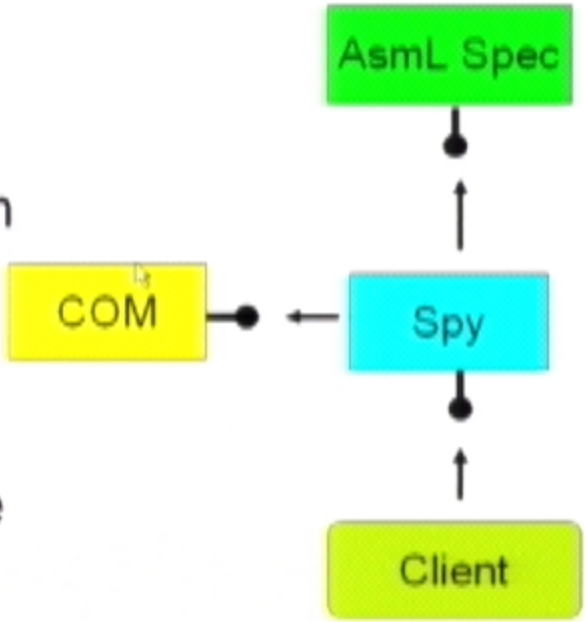
i.e., executable!

# The Big Picture

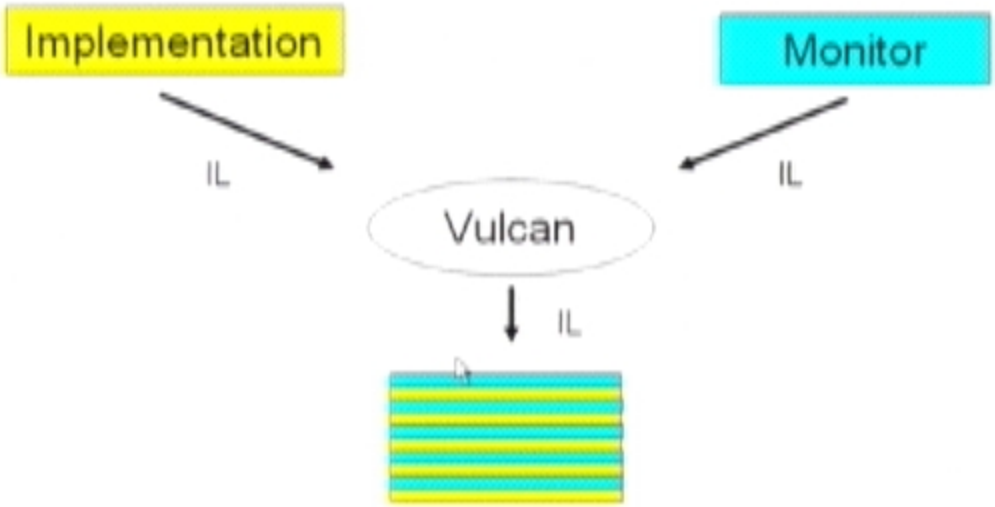


# Previously…

+ Observational equivalence
+ No instrumentation required
- Non-determinism
- Callbacks
- State equivalence



# Tighter Integration: .NET



# (Trivial) Example

```
interface IMath {
  double sqrt(double x);
  ...
}
```

```
class CMath : IMath {
  double sqrt(double x)
    { ... }
  ...
}
```

```
class IMath_Contract : CMath {
  bool sqrt_pre(double x)
    { x >= 0; }
  bool sqrt_post(double x)
    { result * result == x; }
}
```

# Monitored Implementation

```
class CMath {

  double sqrt(double x)
   { double result;
     assert(x >= 0);
     modified body
     assert(result*result == x);
     return result;
   }
}
```

# Monitor Language

```
class I_Contract {
  bool f_pre(...) { }
  bool f_post(...) { }
  bool f_classInvariant(...) { }
  Object f_exception(...) { }
}
```

# Monitor Language

```
class I_Contract {
  bool f_pre(...) { }
  bool f_post(...) { }
  bool f_classInvariant(...) { }
  Object f_exception(...) { }
}
```

# What to specify?

- Properties
  - Pre-/post-conditions
- Calling protocols
  - Required method calls: *mandatory calls*
- Model State
  - Required persistence: *objects*
- Exceptions

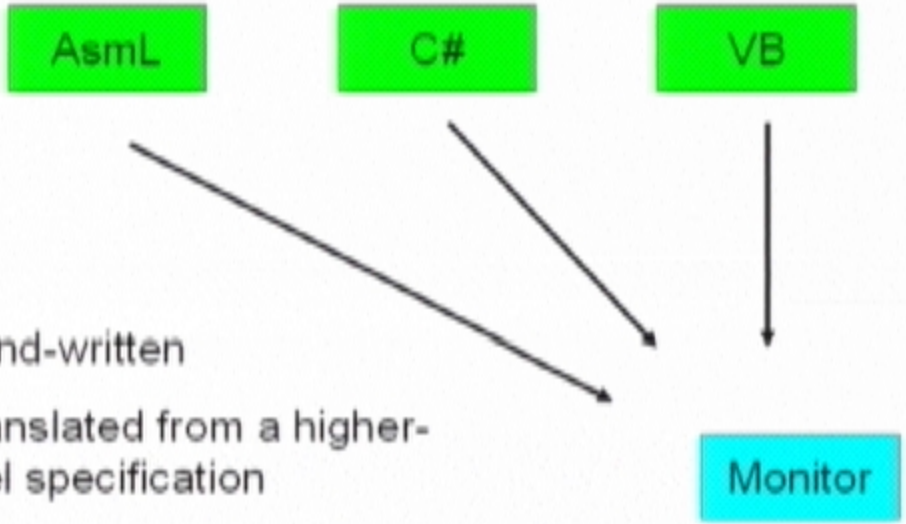
# (Trivial) Example

```
interface IMath {
  double sqrt(double x);
  ...
}
```

```
class CMath : IMath {
  double sqrt(double x)
    { ... }
  ...
}
```

```
class IMath_Contract : CMath {
  bool sqrt_pre(double x)
    { x >= 0; }
  bool sqrt_post(double x)
    { result * result == x; }
}
```

# Monitors


AsmL
C#
VB
•Hand-written
•Translated from a higher-level specification
Monitor

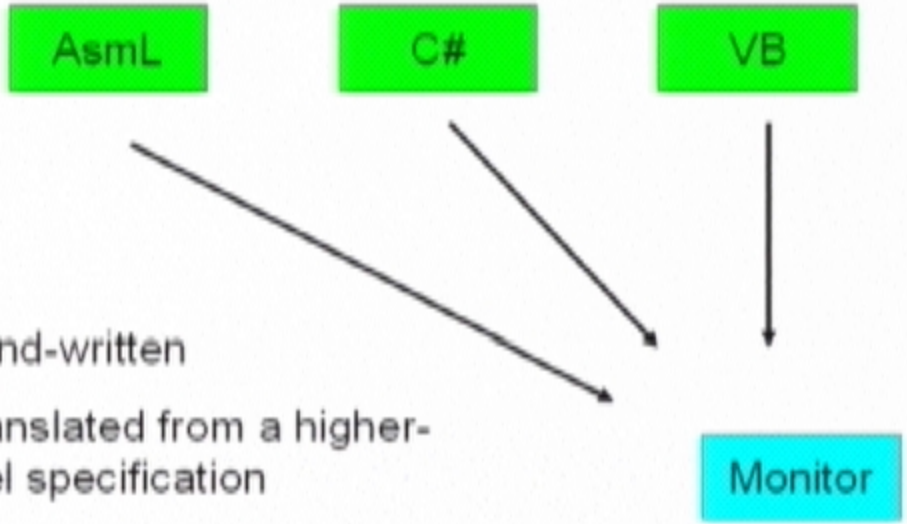
# (Trivial) Example

```
interface IMath {
  double sqrt(double x);
  ...
}
```

```
class CMath : IMath {
  double sqrt(double x)
    { ... }
  ...
}
```

```
class IMath_Contract : CMath {
  bool sqrt_pre(double x)
    { x >= 0; }
  bool sqrt_post(double x)
    { result * result == x; }
}
```

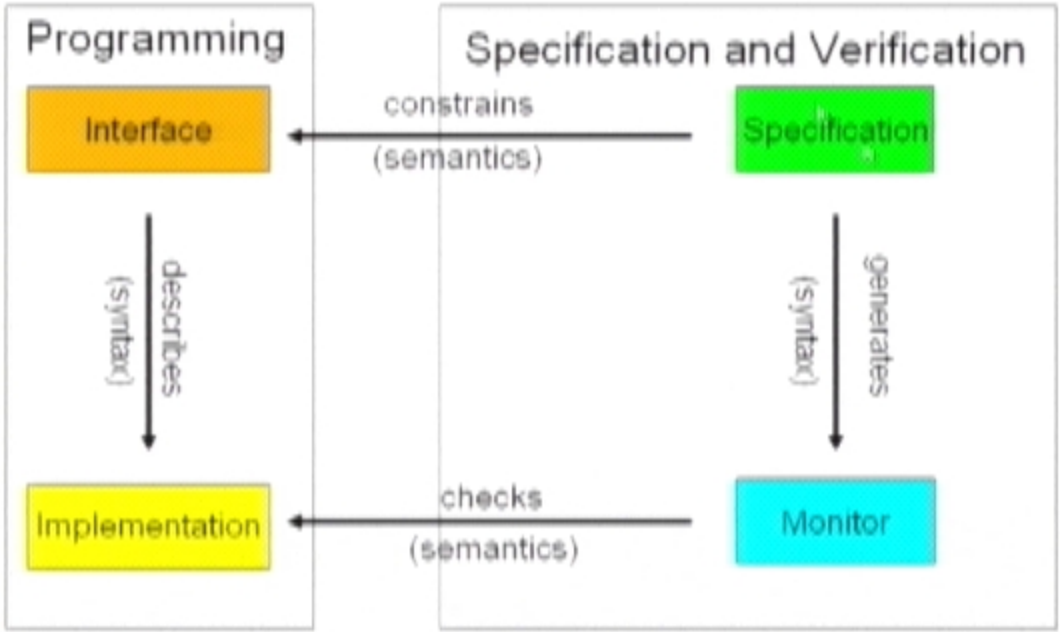
# Monitors


AsmL
C#
VB
•Hand-written
•Translated from a higher-level specification
Monitor

# AsmL: Microsoft's ASM-Language

- A powerful specification language:
  - Based on ASMs
  - Readable, executable, testable
  - Combines mathematical, object-oriented, component-oriented approaches
  - Integrated into VS6 (working on VS.NET)
  - Literate style: Word and XML

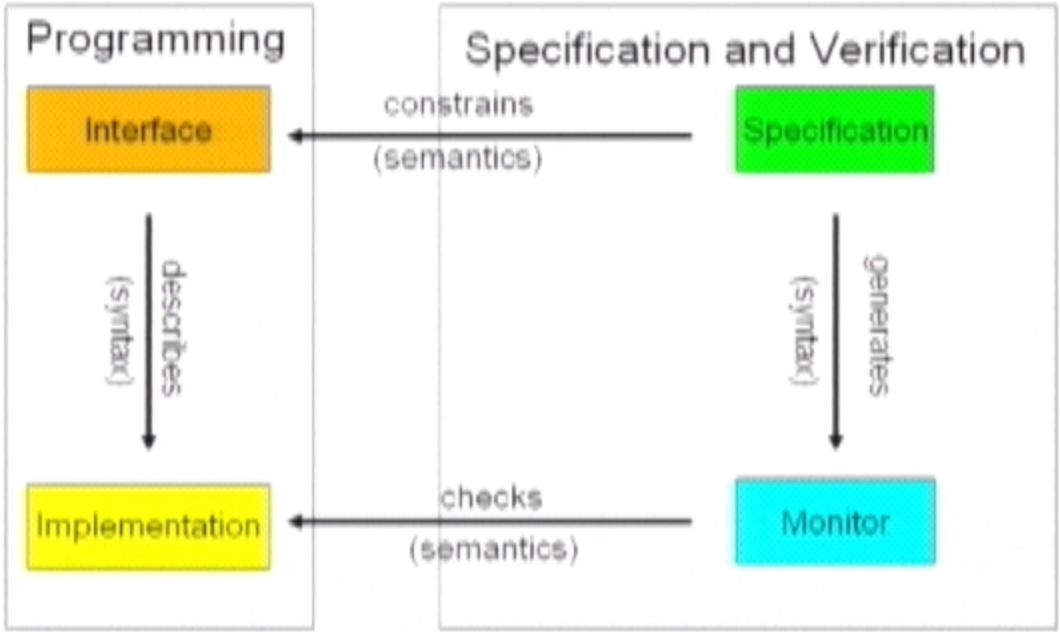
# The Big Picture



# AsmL: Microsoft's ASM-Language

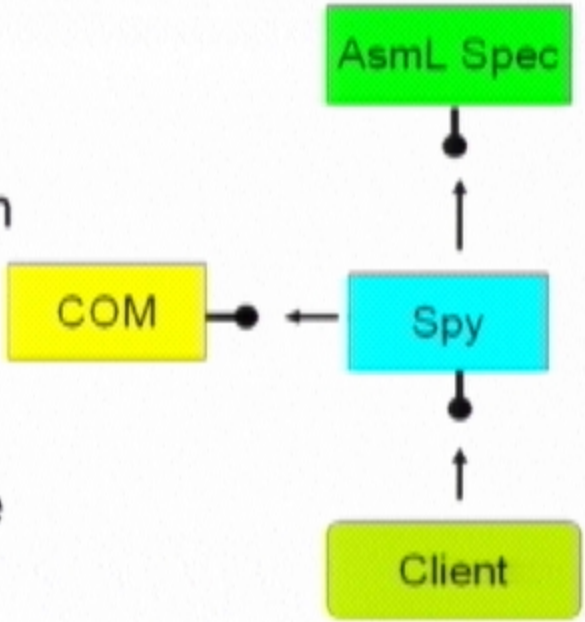
- A powerful specification language:
  - Based on ASMs
  - Readable, executable, testable
  - Combines mathematical, object-oriented, component-oriented approaches
  - Integrated into VS6 (working on VS.NET)
  - Literate style: Word and XML

# The Big Picture



# Previously…

+ Observational equivalence
+ No instrumentation required
- Non-determinism
- Callbacks
- State equivalence



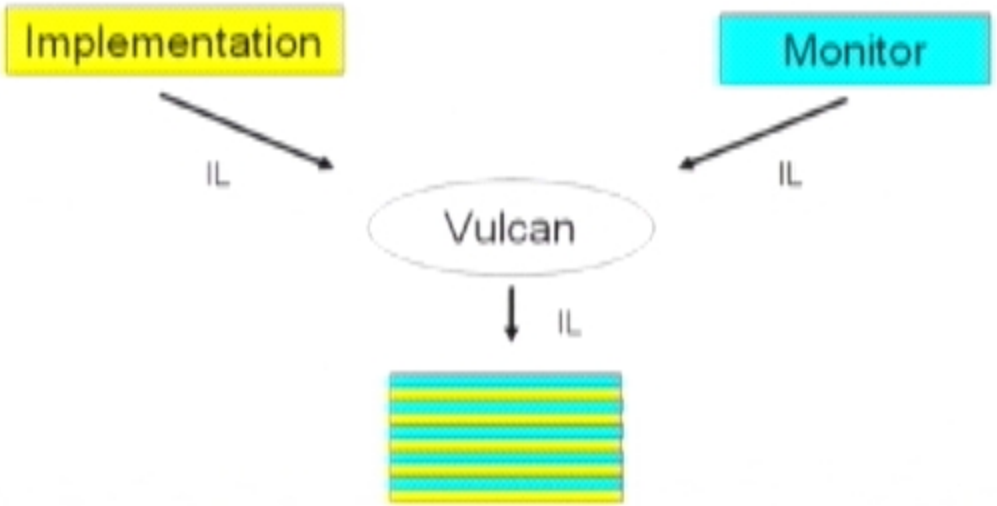
# (Trivial) Example

```
interface IMath {
  double sqrt(double x);
  ...
}
```

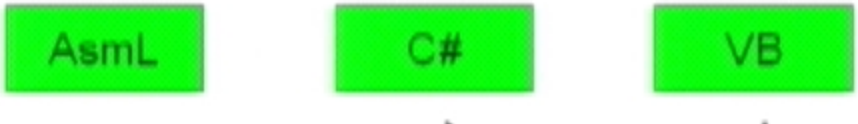
```
class CMath : IMath {
  double sqrt(double x)
    { ... }
  ...
}
```

```
class IMath_Contract : CMath {
  bool sqrt_pre(double x)
    { x >= 0; }
  bool sqrt_post(double x)
    { result * result == x; }
}
```

# Tighter Integration: .NET



# Monitors




- Hand-written
- Translated from a higher-level specification

Monitor

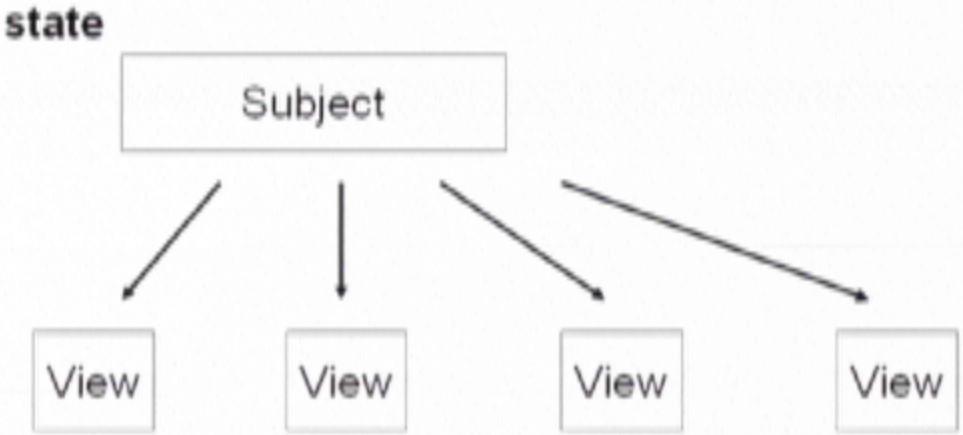
# (Trivial) Example

```
interface IMath {
  double sqrt(double x);
  ...
}
```

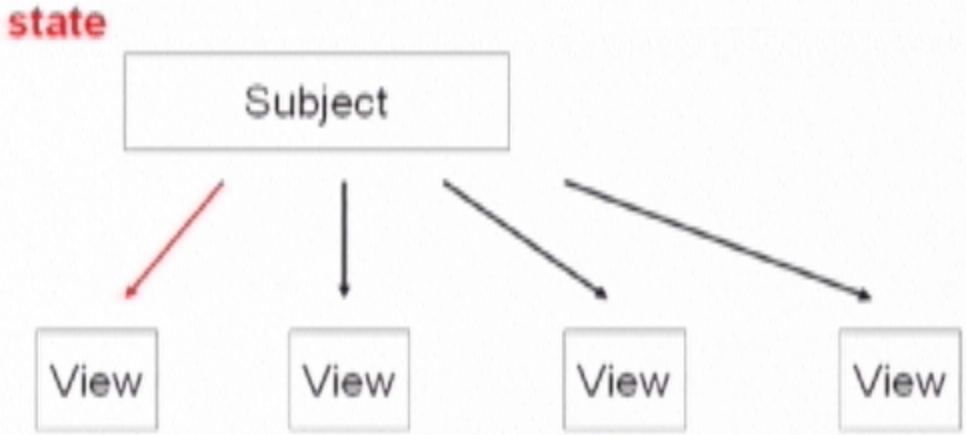
```
class CMath : IMath {
  double sqrt(double x)
    { ... }
  ...
}
```

```
class IMath_Contract : CMath {
  bool sqrt_pre(double x)
    { x >= 0; }
  bool sqrt_post(double x)
    { result * result == x; }
}
```

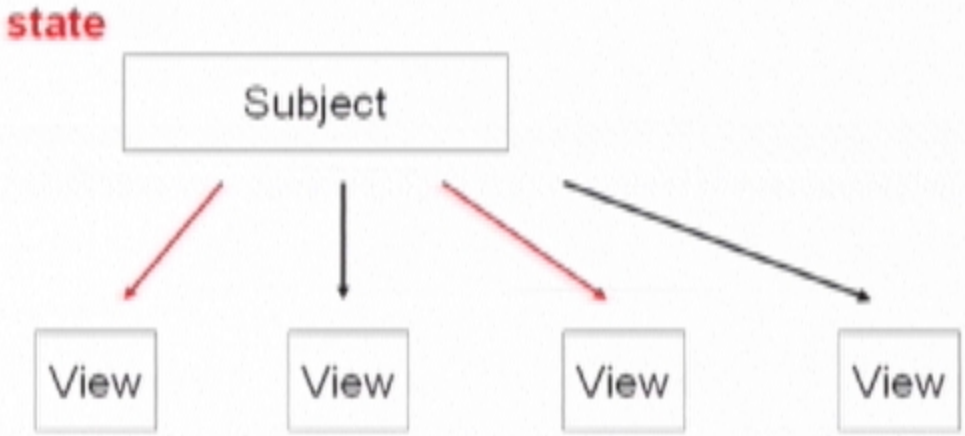
# Subject/View



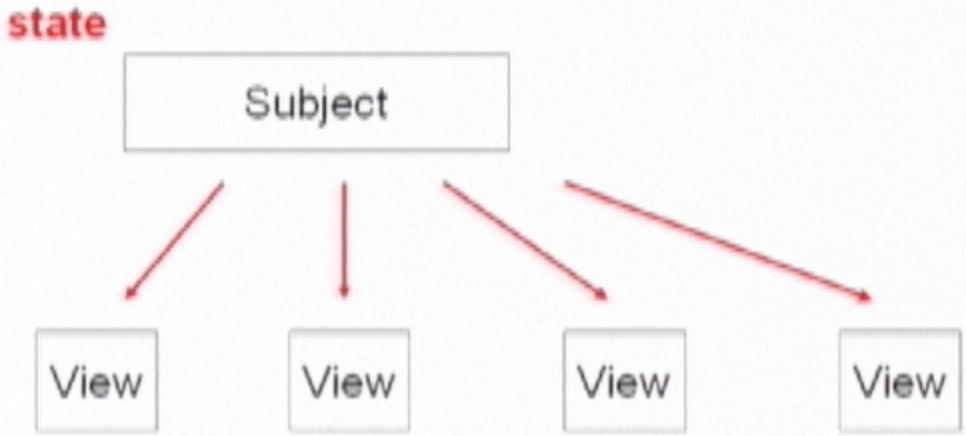
# Subject/View



# Subject/View



# Subject/View



# Subject/View in AsmL

```
class Subject[T]
  state as T
  views as Set[IView]

  Get() as T =  state

  Set(x as T) =
    machine
      state := x
    step
      forall v in views do
        v.Update()
```

# Subject/View in AsmL

```
class Subject[T]
  state as T
  views as Set[IView]

  Get() as T =  state

  Set(x as T) =
    machine
      state := x
    step
      forall v in views do
        v.Update()
```

Notify **after** update

Parallel **forall:** all views see same state

# Subject/View in AsmL

```
class Subject[T]
  state as T
  views as Set[IView]

  Get() as T =  state

  Set(x as T) =
    machine
      state := x
    step
      forall v in views do
        v.Update()
```

Notify **after** update

Parallel **forall:** all views see same state

Mandatory call: once per view

# Deriving the Monitor from AsmL

```
Set(x as T) =
  machine  state := x
  step     forall v in views do v.Update()
```

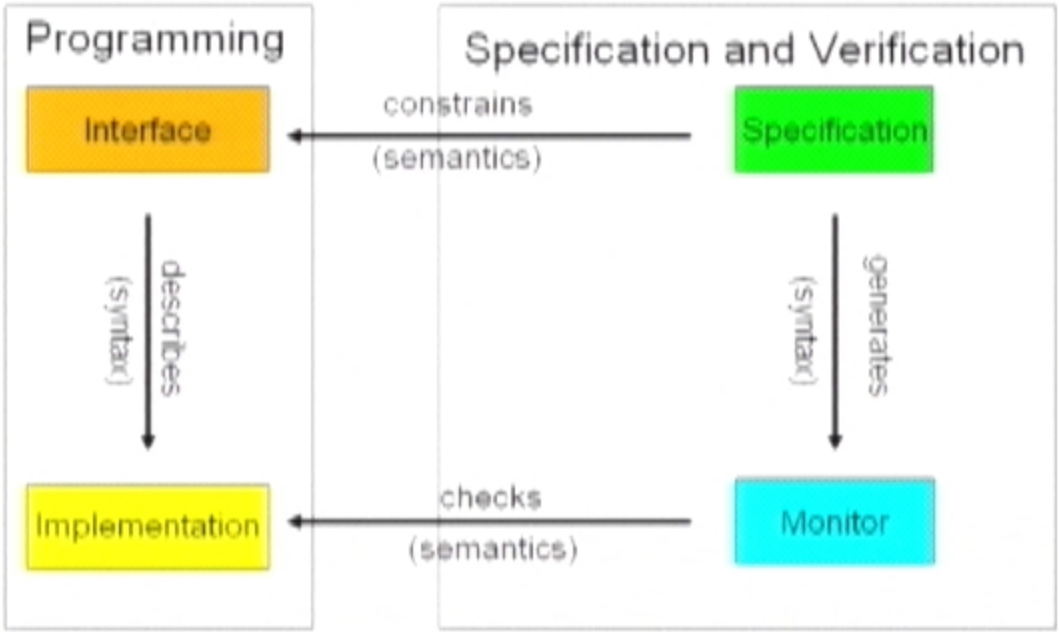
```
Set_pre() { mcalls = { v.Update | v in views }; }
Set_body() { state = x; }
Set_post() { assert(mcalls == {}); }

Update_pre() { assert(state == x); }
Update_post() { mcalls.remove(v.Update) }
```

# Summary: Runtime Verification

- Monitor design can be used for arbitrary purposes
- Monitor is generated automatically from AsmL
  - Can be added incrementally
  - Does not require model checking, theorem proving, etc.

http://asml

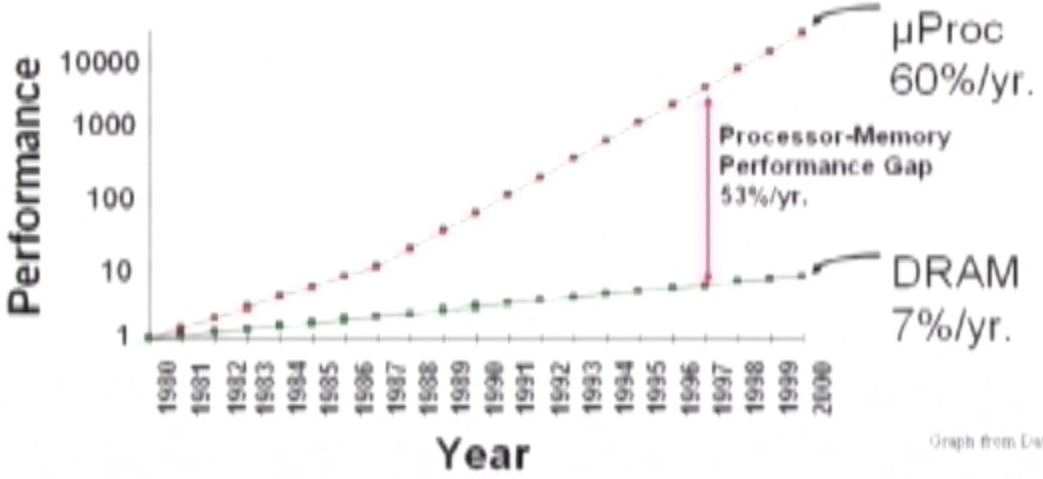
# The Big Picture



# Using Garbage Collection to Improve Program Data Locality

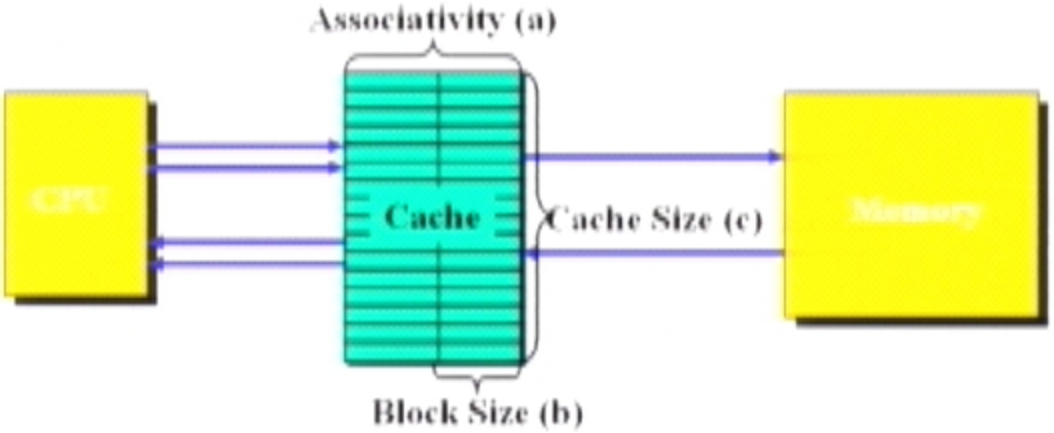
Trishul Chilimbi
PPRC, Performance Monitoring & Analysis
MS Research
http://research.microsoft.com/~trishulc/Daedalus.htm

# Processor-Memory Imbalance

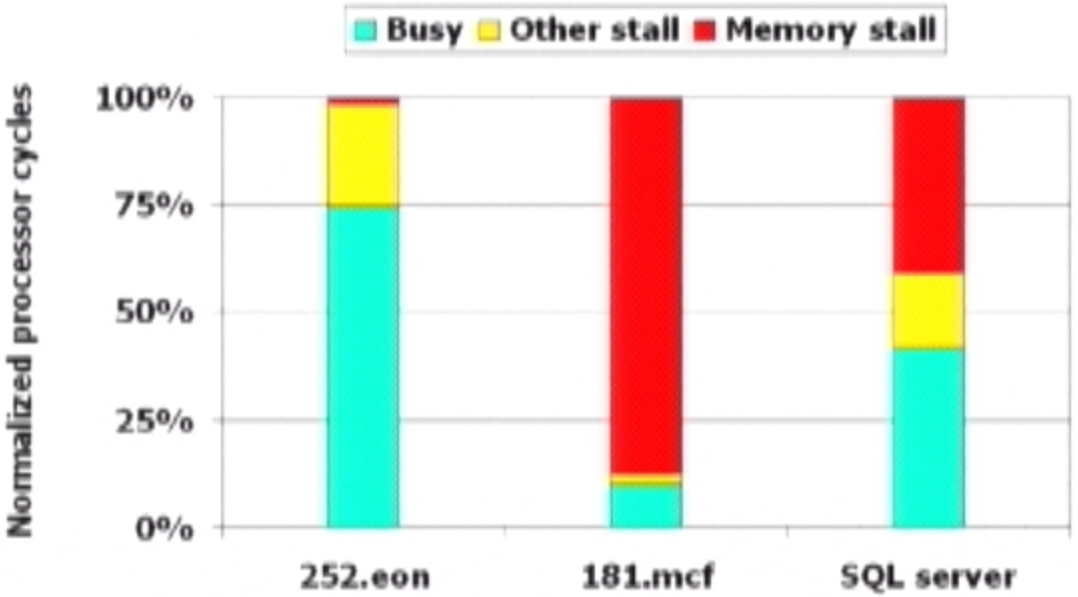



Graph from Dave Patterson

# Memory Cache: Reduce Latency



# Performance Impact



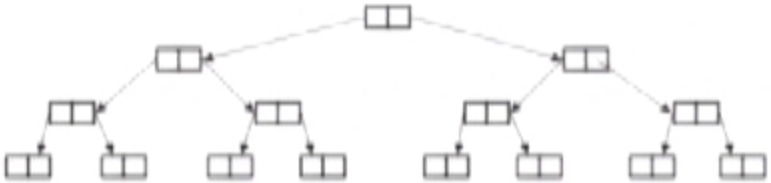
# Talk Outline

- Background
- Motivation
- GC for implementing cache-conscious data layouts
- Low-overhead data reference profiling
- Fast and accurate data layout determination

# Improving Memory System Performance

- New Hardware
  - Better caches
- New Software
  - Redesign code and structures
  - Cache-conscious algorithms
- Our approach: Improve locality of existing software automatically
  - Cache-conscious data layouts

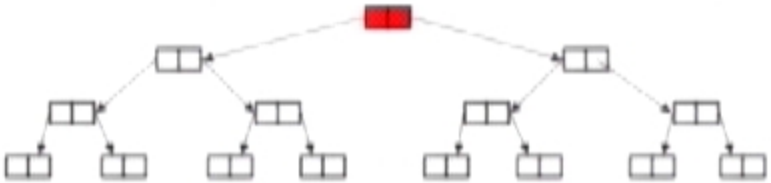
# Cache-Conscious Data Layout



Random Layout

Cache Conscious

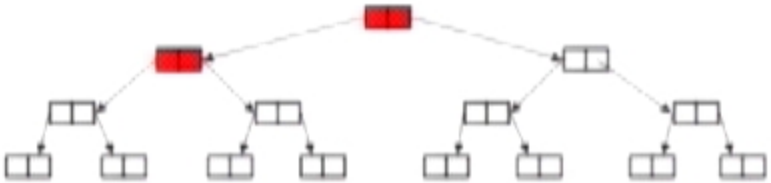
# Cache-Conscious Data Layout



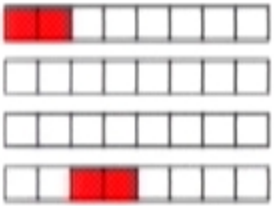
Random Layout

Cache Conscious

# Cache-Conscious Data Layout

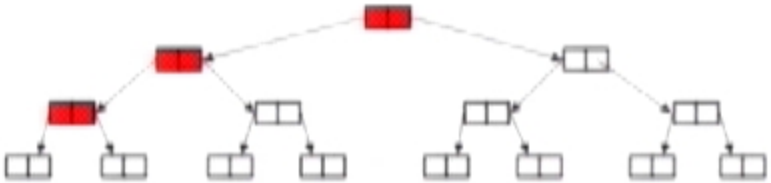


Random Layout

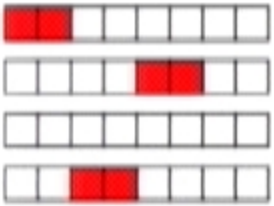


Cache Conscious

# Cache-Conscious Data Layout

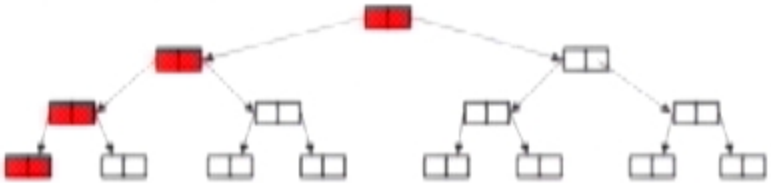


Random Layout

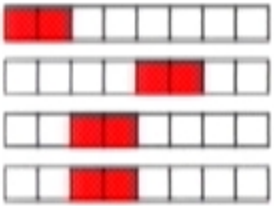


Cache Conscious

# Cache-Conscious Data Layout

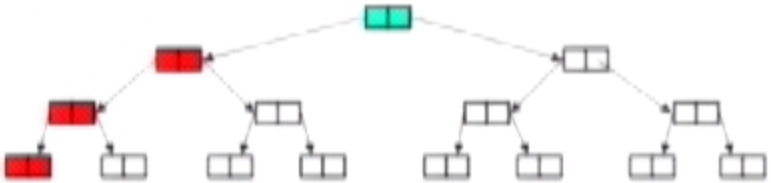


Random Layout

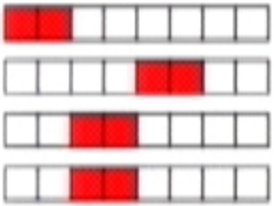


Cache Conscious

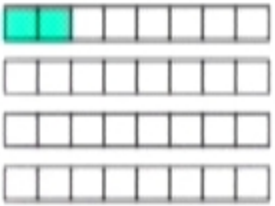
# Cache-Conscious Data Layout



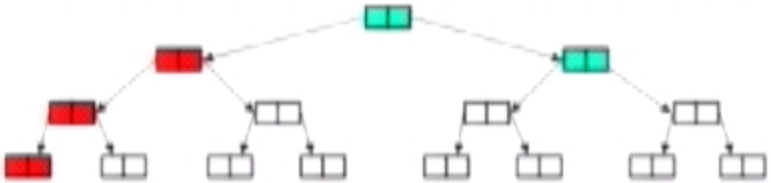
Random Layout



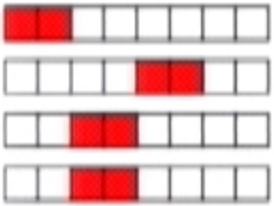
Cache Conscious



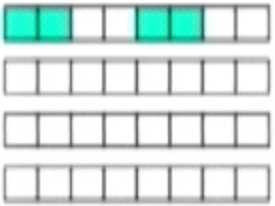
# Cache-Conscious Data Layout



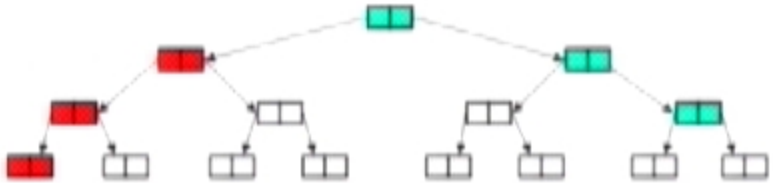
Random Layout



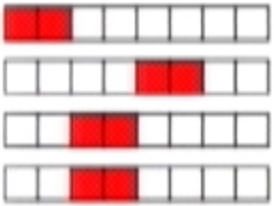
Cache Conscious



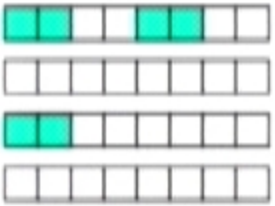
# Cache-Conscious Data Layout



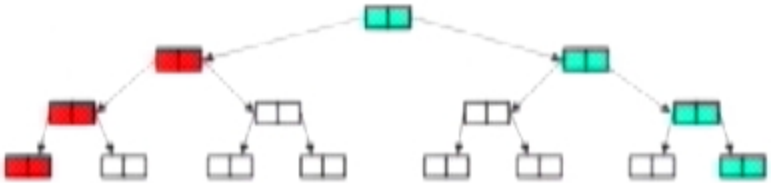
Random Layout



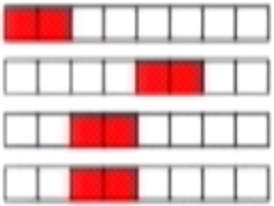
Cache Conscious



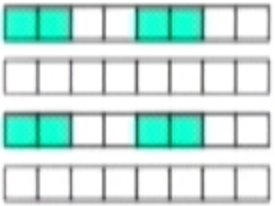
# Cache-Conscious Data Layout



Random Layout



Cache Conscious



# Talk Outline

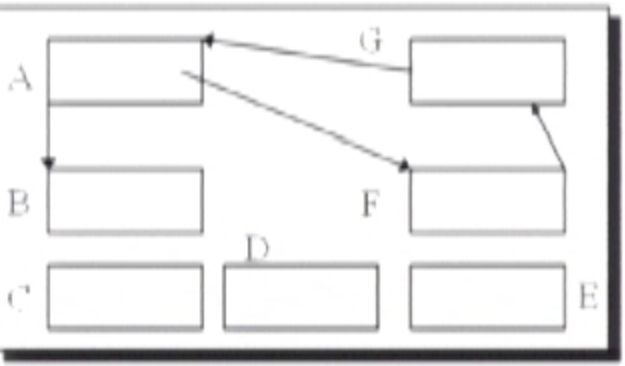
- Background and Motivation
- GC for implementing cache-conscious data layouts
  - Structure reorganization
  - Hot/cold splitting
  - Limitations
- Low-overhead data reference profiling
- Fast and accurate data layout determination

# Copying Garbage Collection

FROM space

TO space

Roots: A, C, E

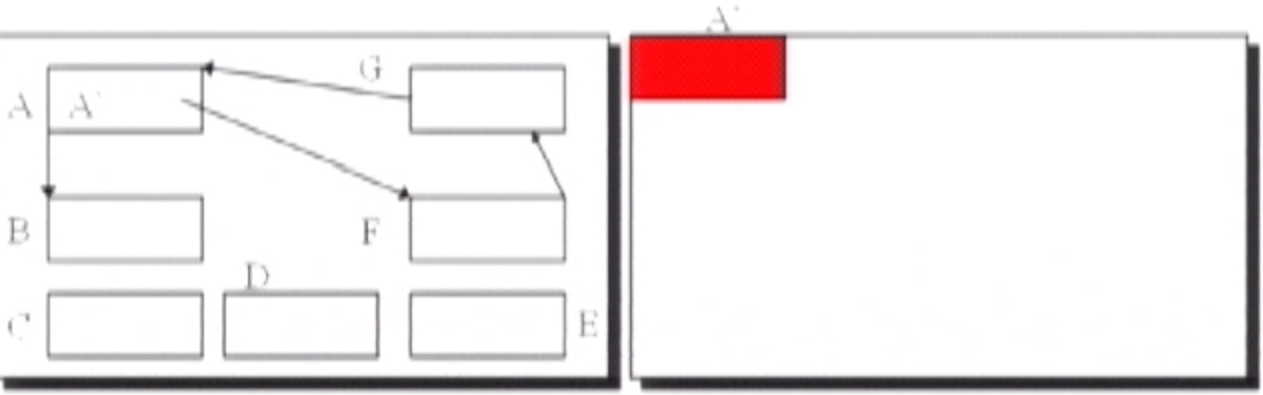


# Copying Garbage Collection

FROM space

TO space

Roots: A, C, E

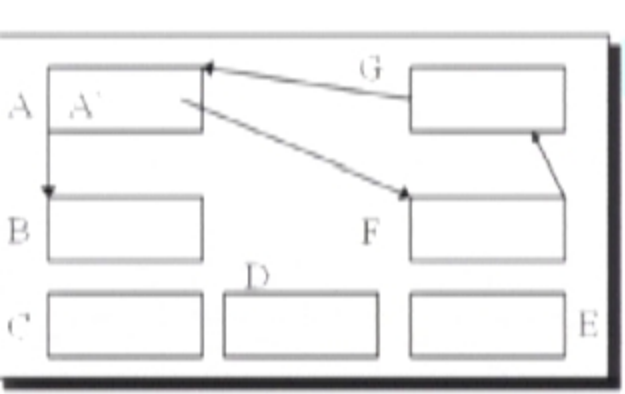


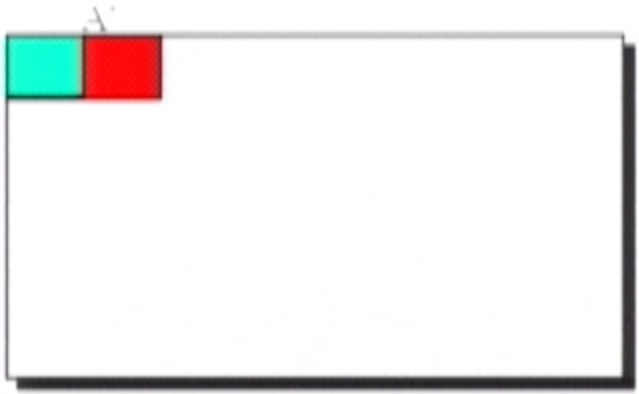
# Copying Garbage Collection

FROM space

TO space

Roots: A, C, E



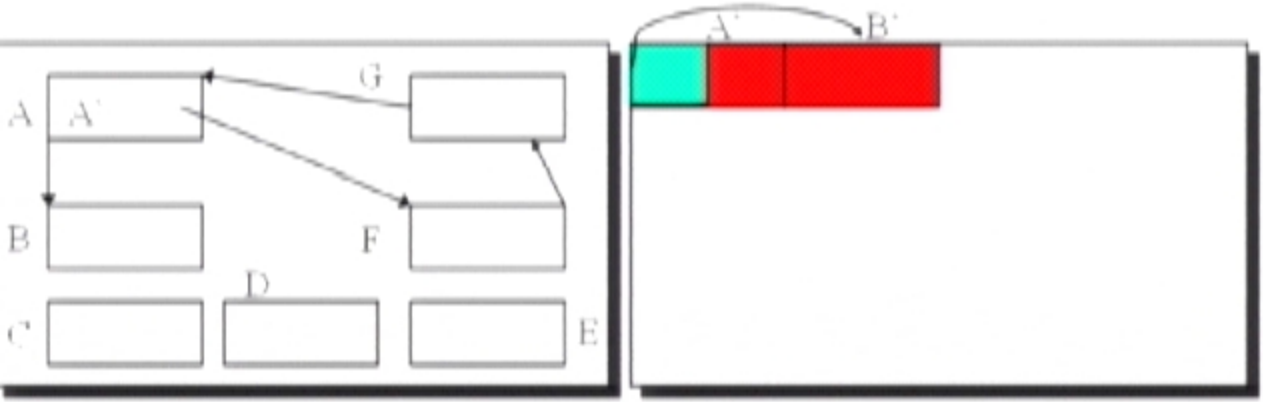



# Copying Garbage Collection

FROM space

TO space

Roots: A, C, E

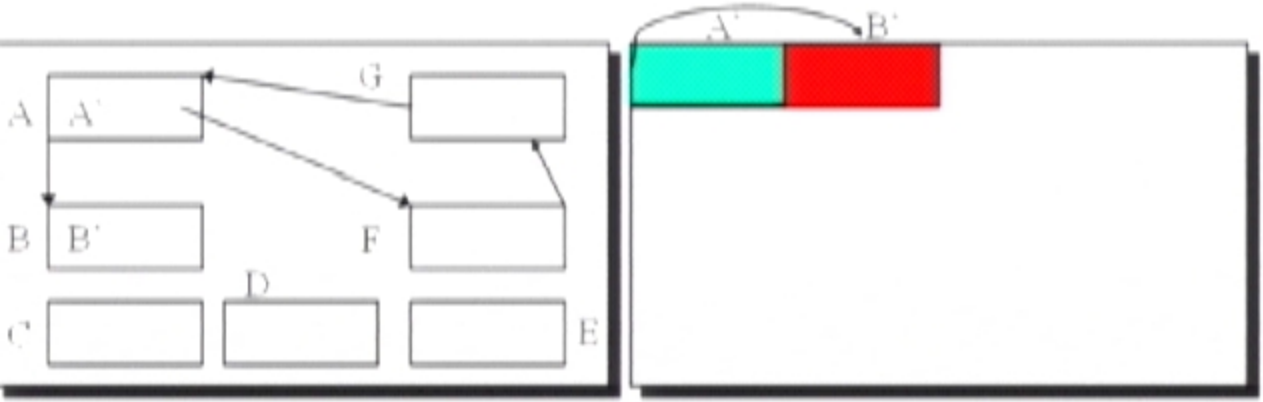


# Copying Garbage Collection

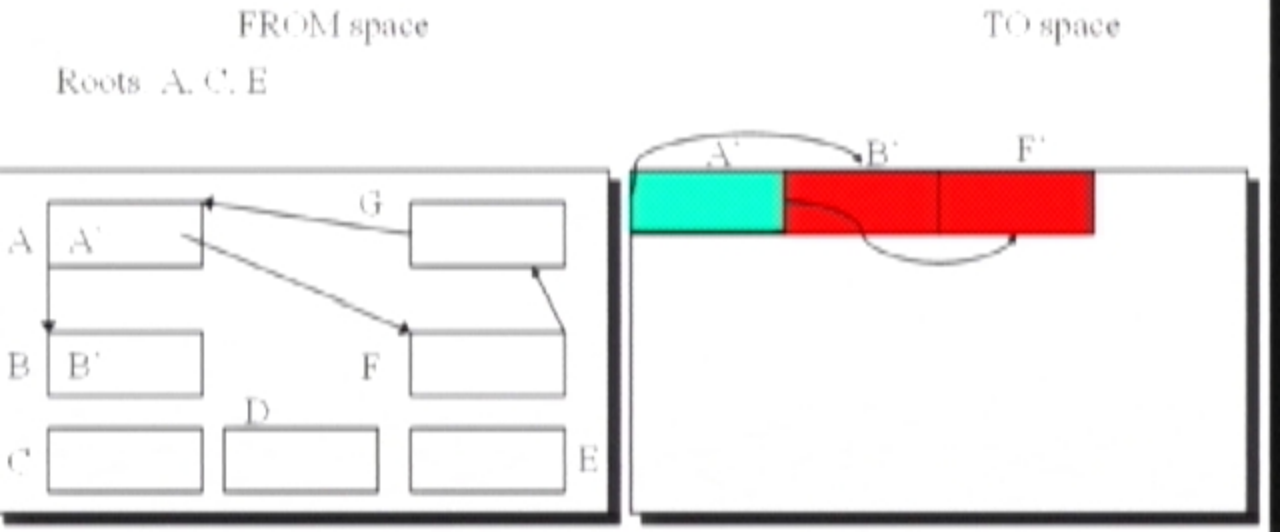
FROM space

TO space

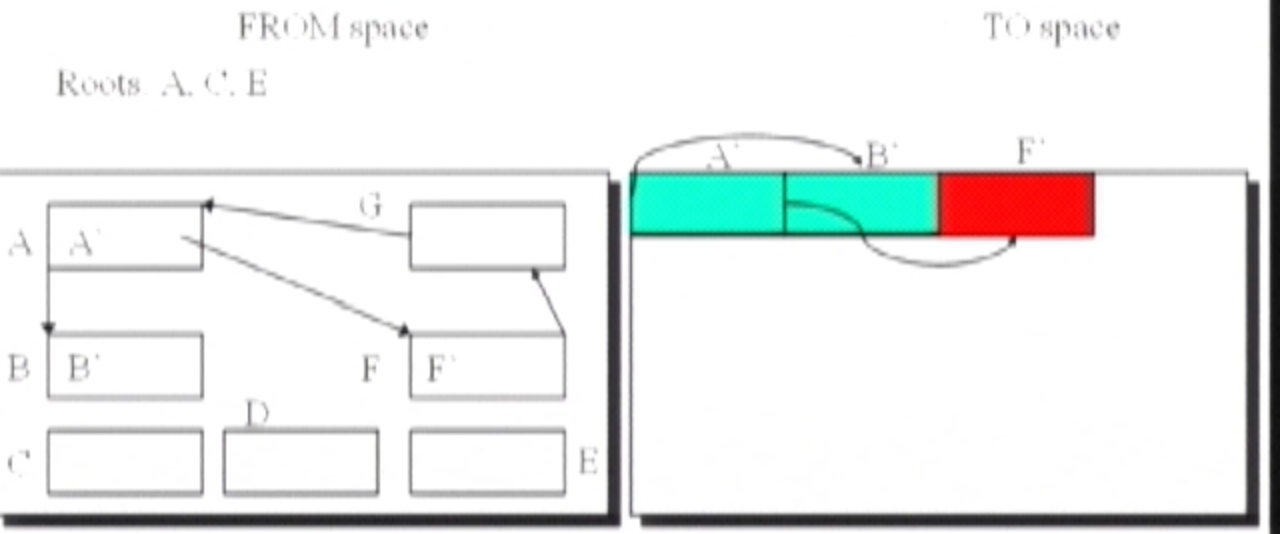
Roots: A, C, E



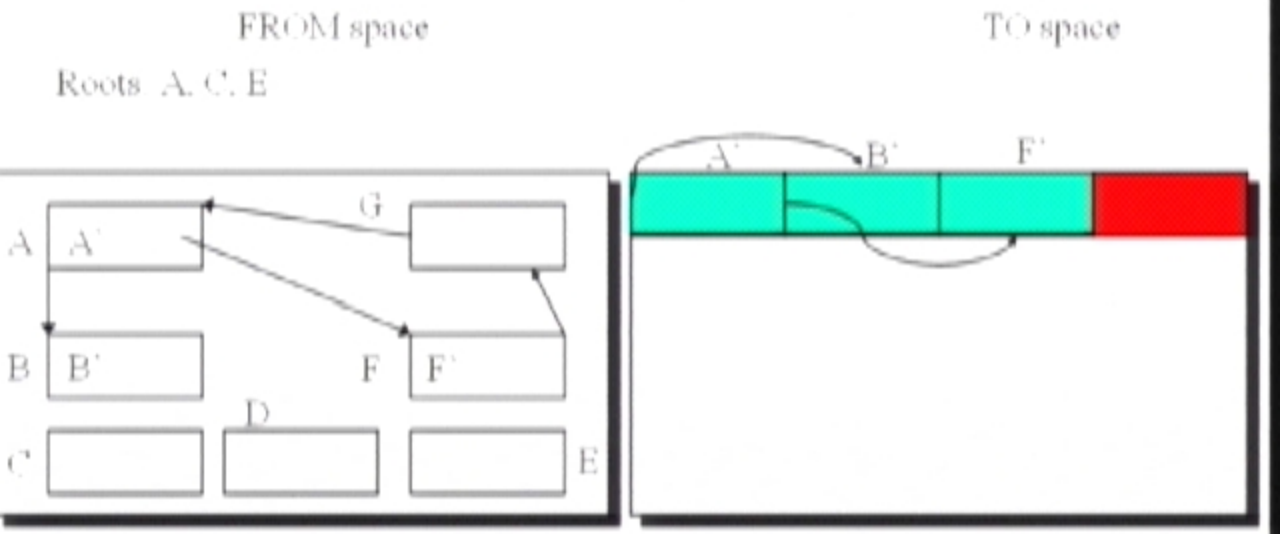
# Copying Garbage Collection



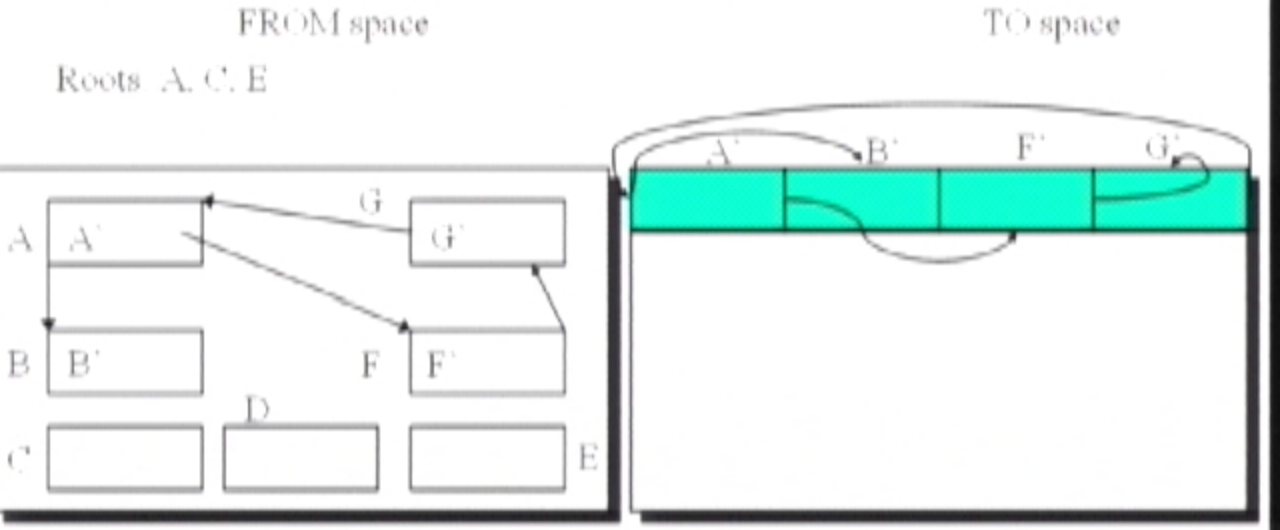
# Copying Garbage Collection



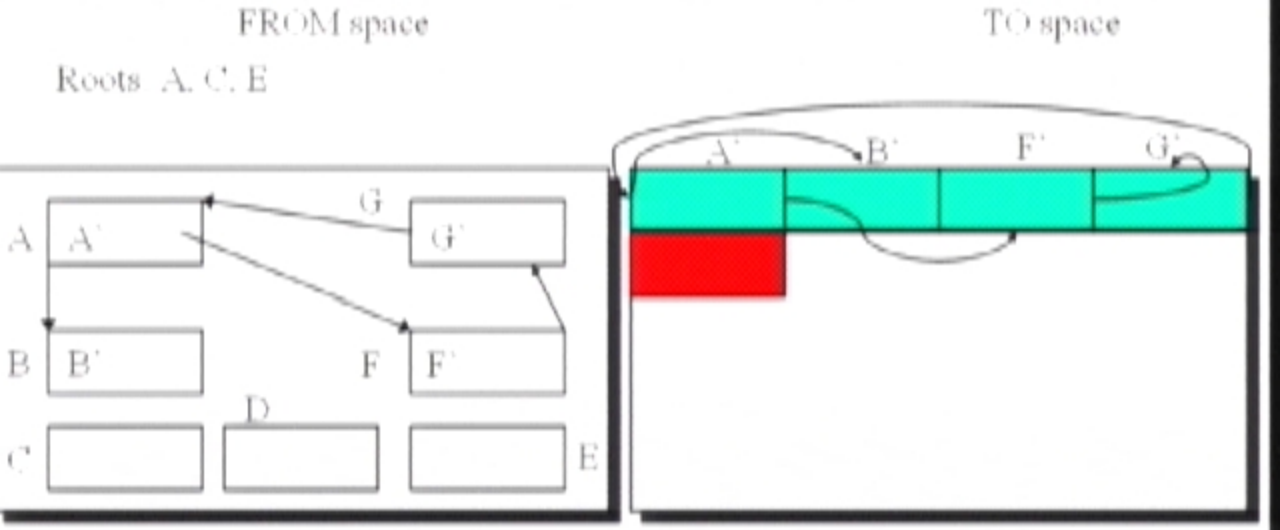
# Copying Garbage Collection



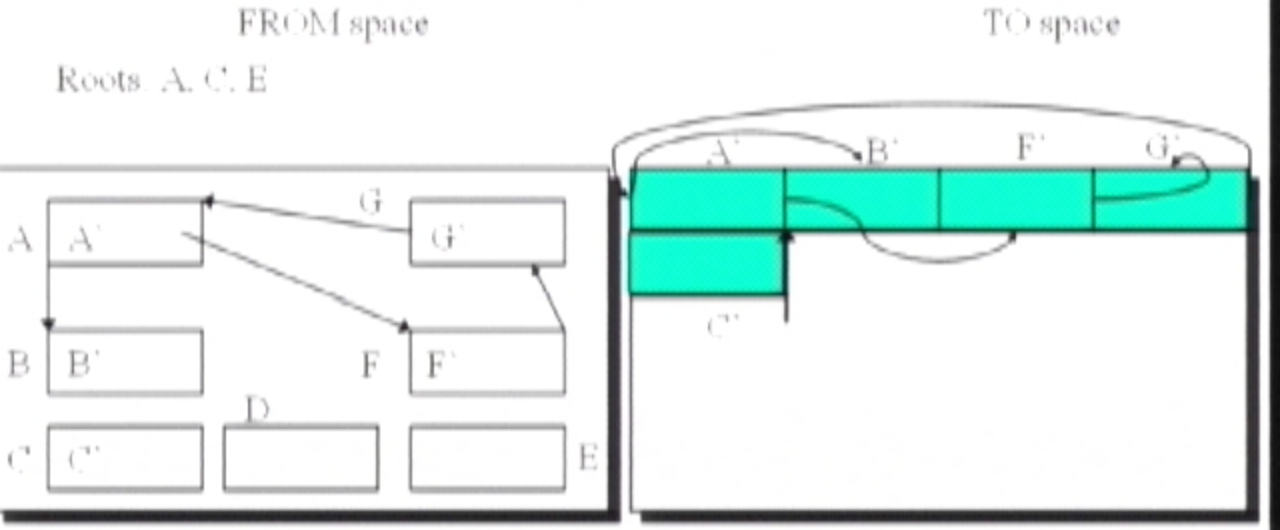
# Copying Garbage Collection



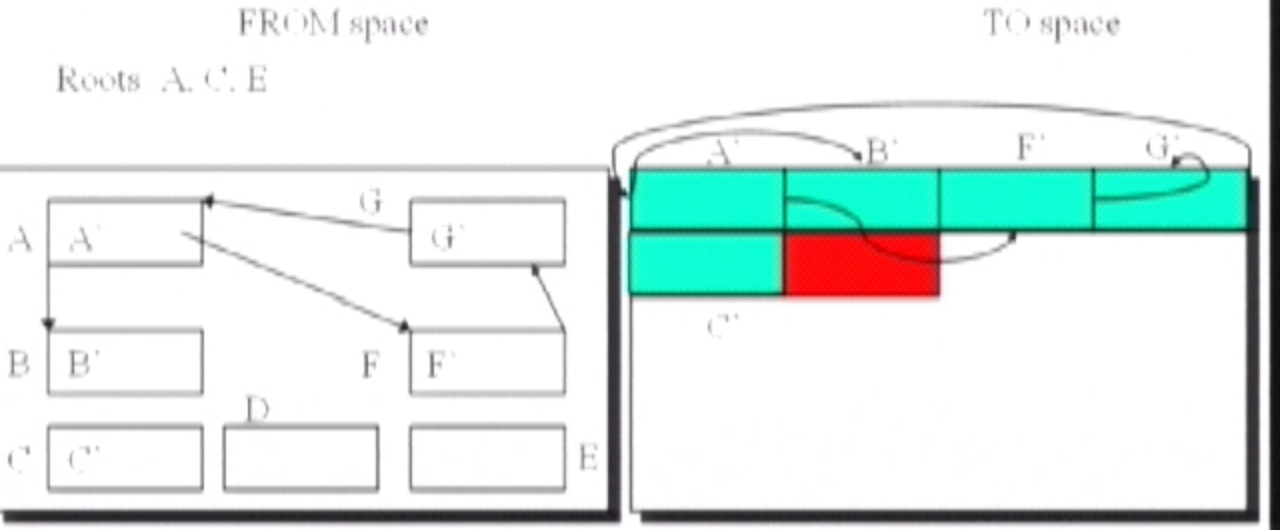
# Copying Garbage Collection



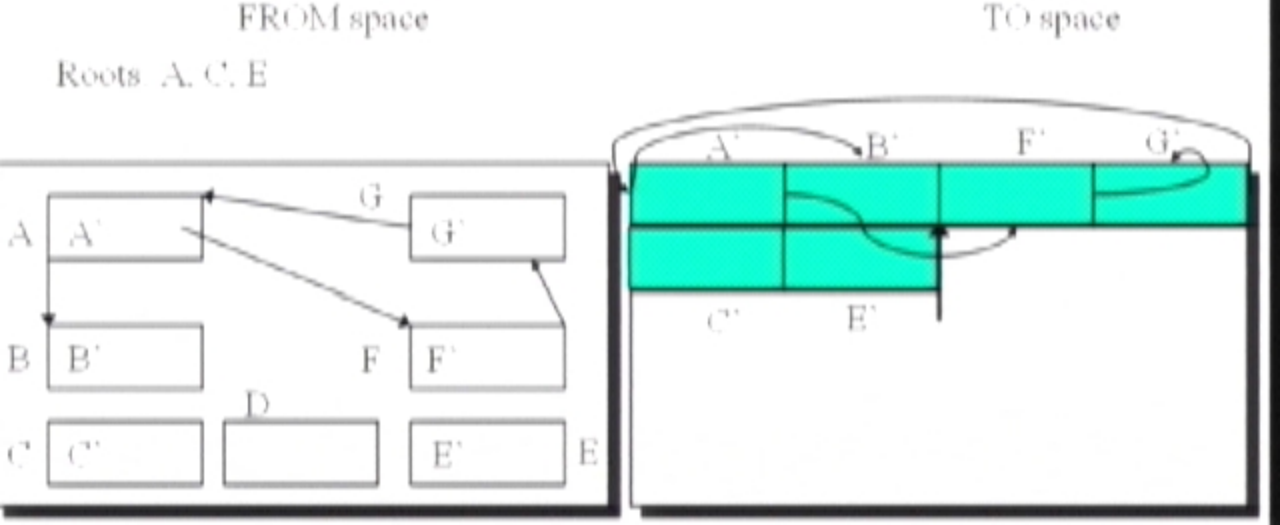
# Copying Garbage Collection



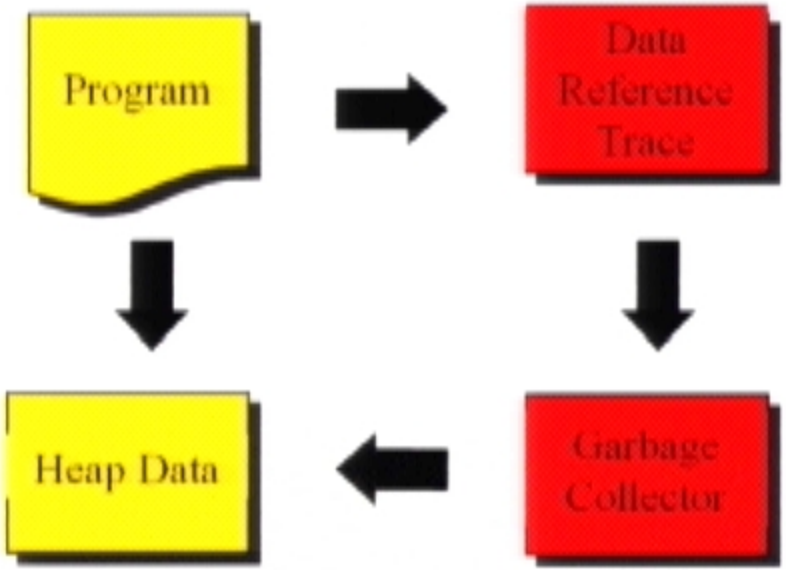
# Copying Garbage Collection



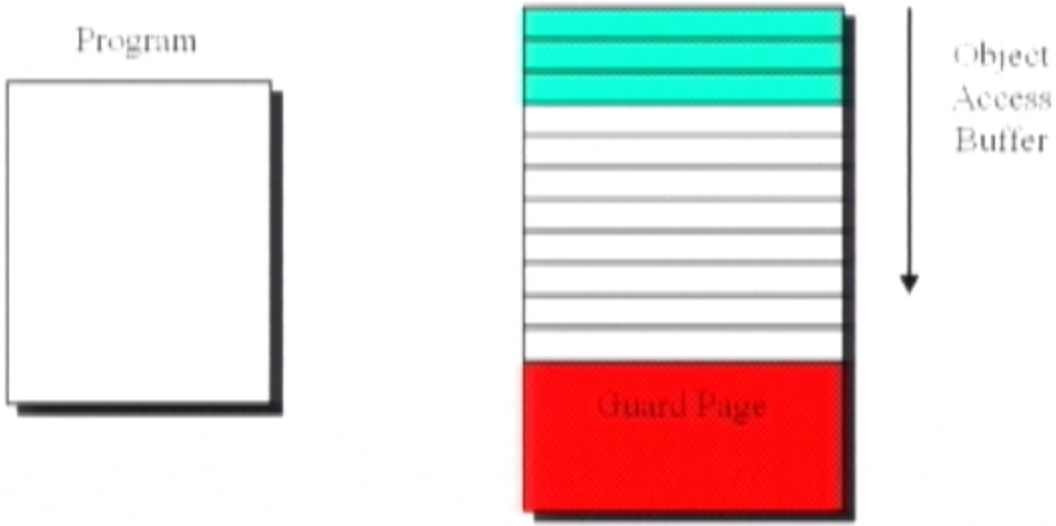
# Copying Garbage Collection



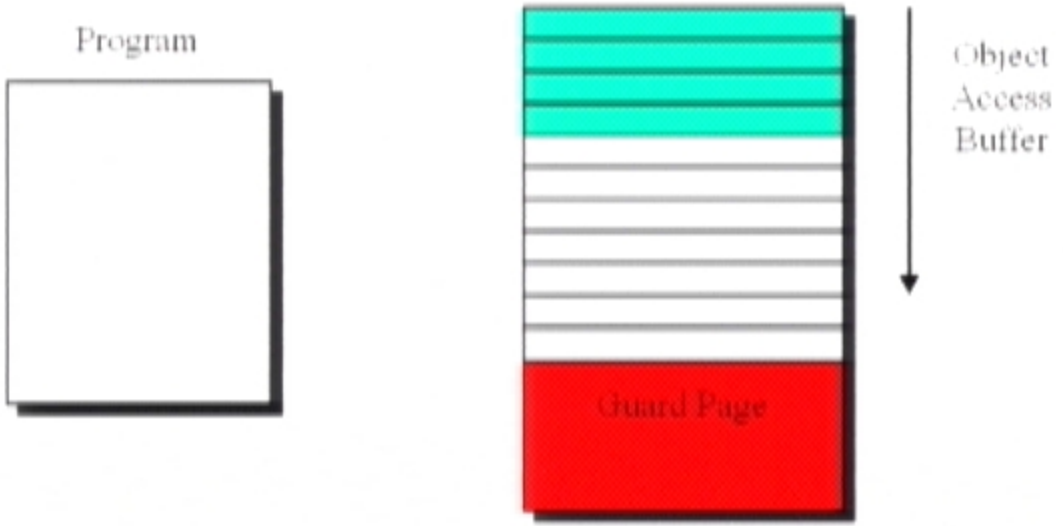
# Using Copying GC for Cache-Conscious Reorganization



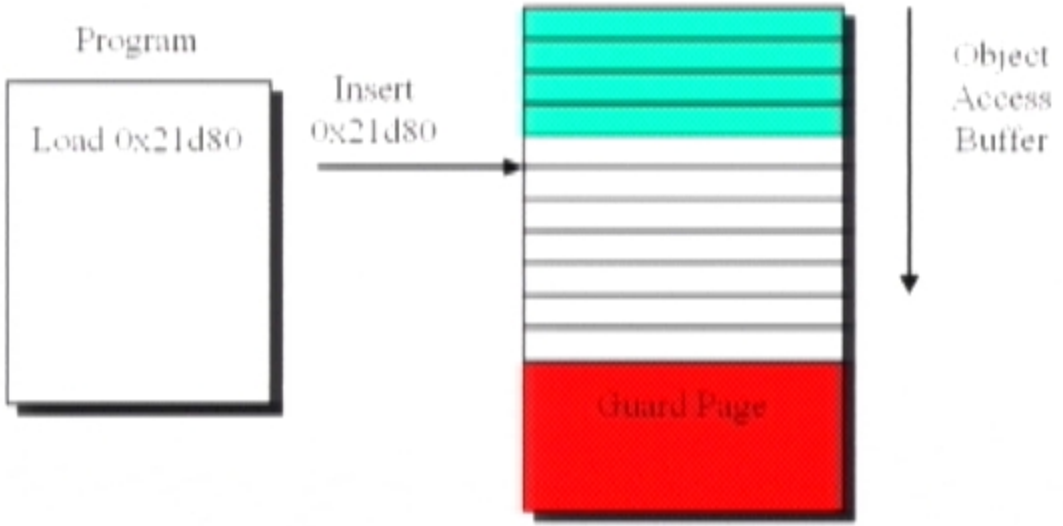
# Object Access Buffer



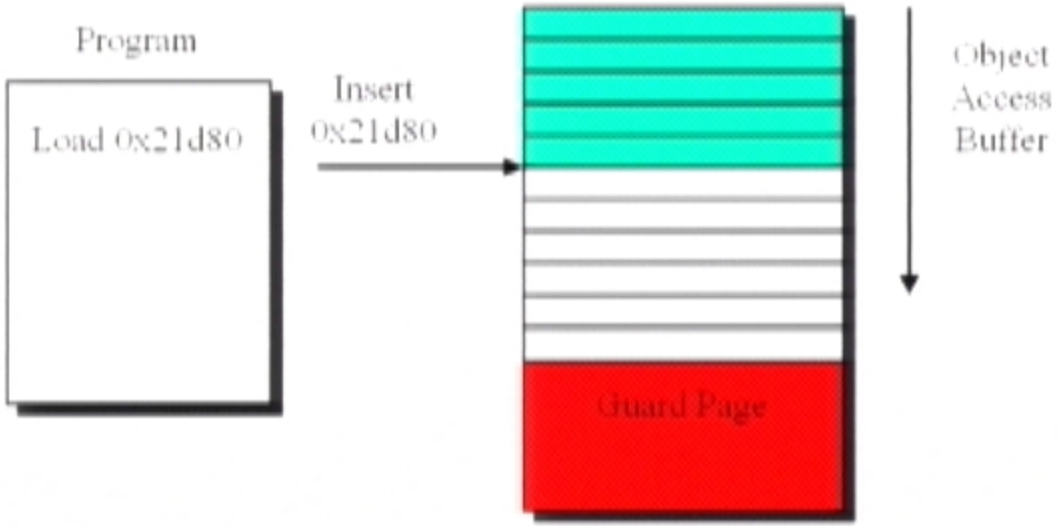
# Object Access Buffer



# Object Access Buffer



# Object Access Buffer

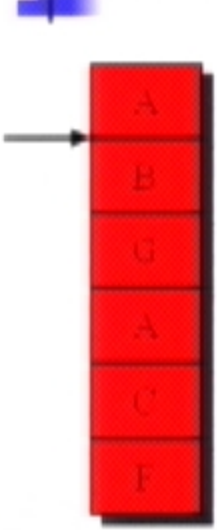


# Constructing the Object Affinity Graph


Object Access Buffer

Locality Queue

# Constructing the Object Affinity Graph



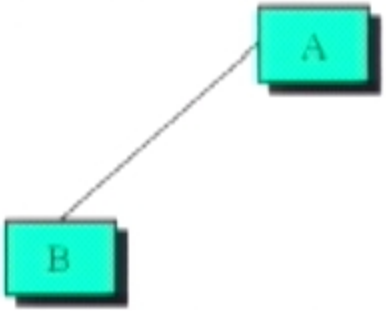


Object Affinity Graph

Object Access Buffer

Locality Queue

# Constructing the Object Affinity Graph

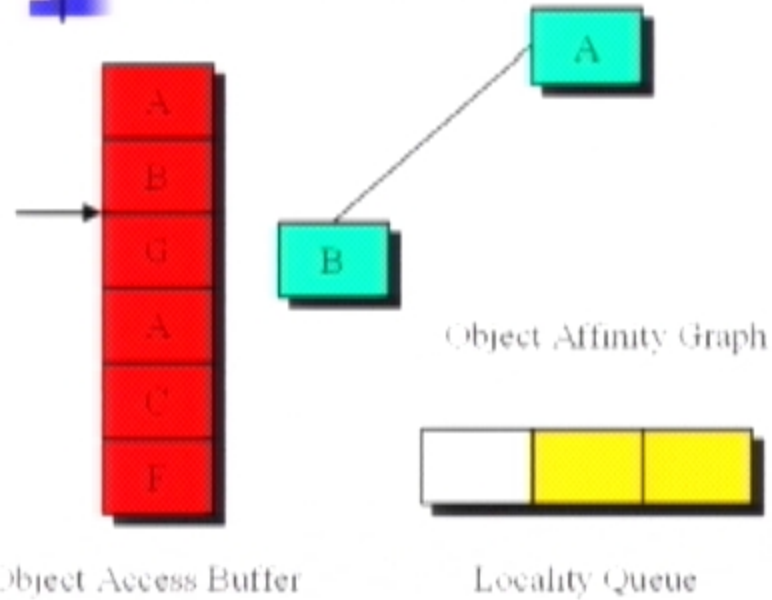




Object Affinity Graph

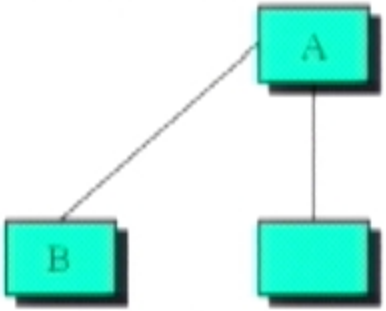
Locality Queue

Object Access Buffer

# Constructing the Object Affinity Graph



# Constructing the Object Affinity Graph

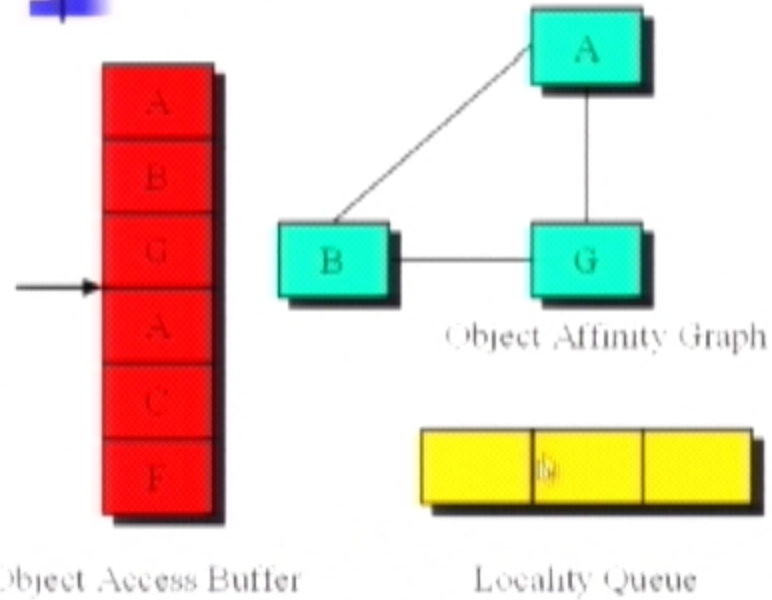




Object Affinity Graph

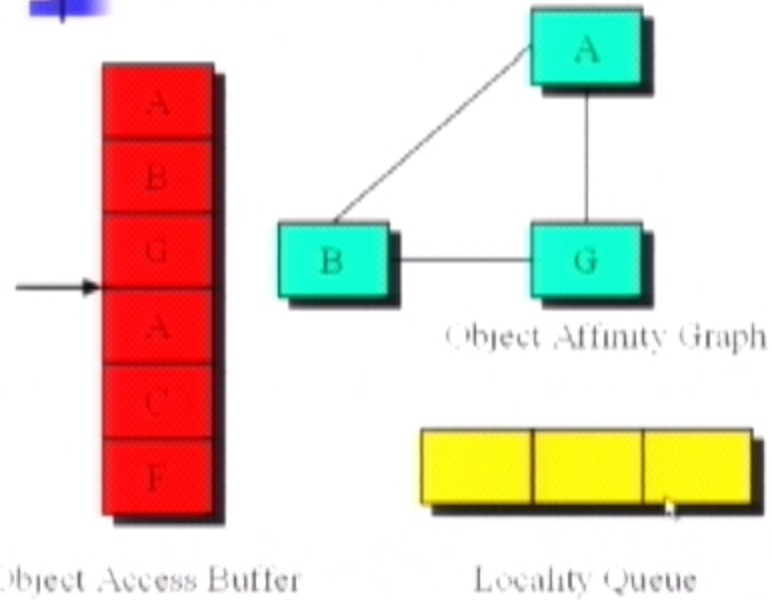
Object Access Buffer

Locality Queue

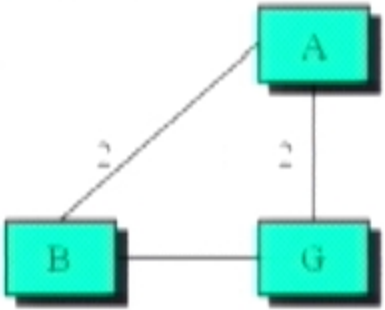
# Constructing the Object Affinity Graph



# Constructing the Object Affinity Graph



# Constructing the Object Affinity Graph

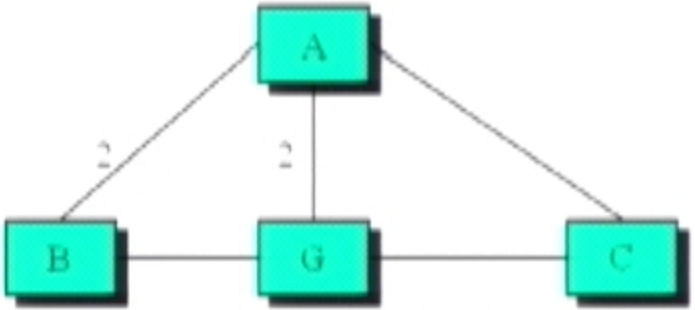




Object Affinity Graph

Object Access Buffer

Locality Queue

# Constructing the Object Affinity Graph

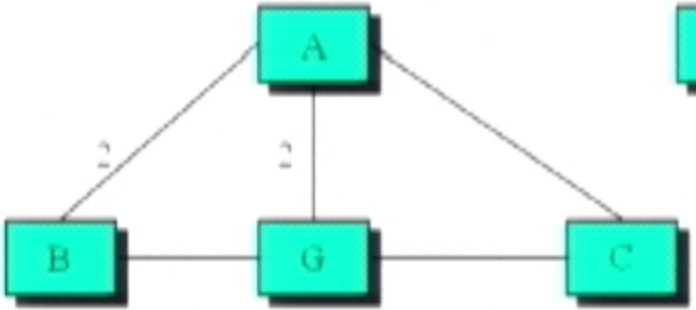




Object Affinity Graph

Object Access Buffer

Locality Queue

# Constructing the Object Affinity Graph

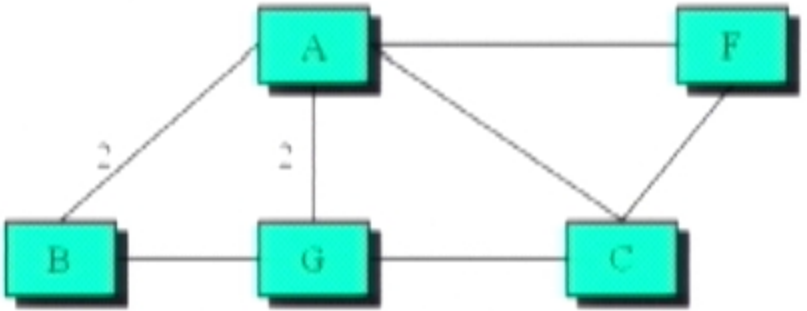

Object Affinity Graph

Object Access Buffer

Locality Queue

# Constructing the Object Affinity Graph

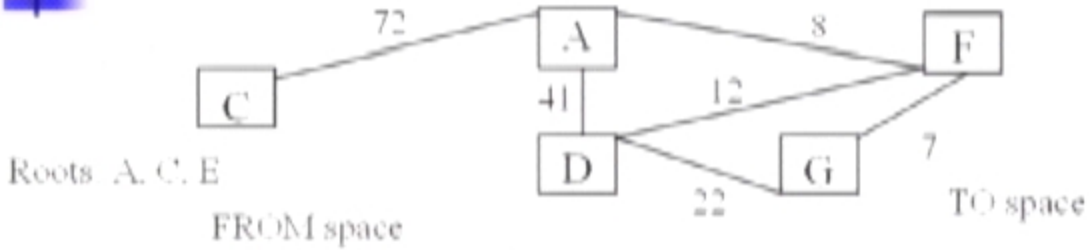

Object Access Buffer

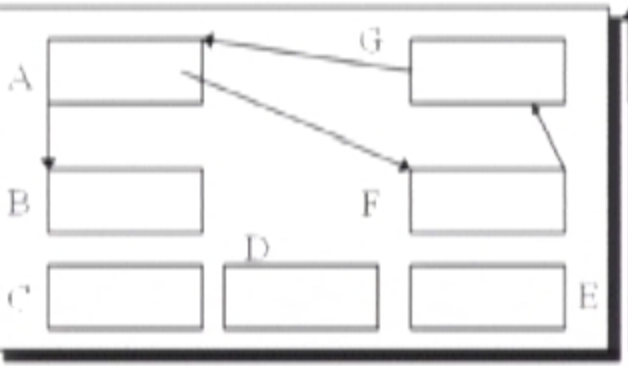



Object Affinity Graph

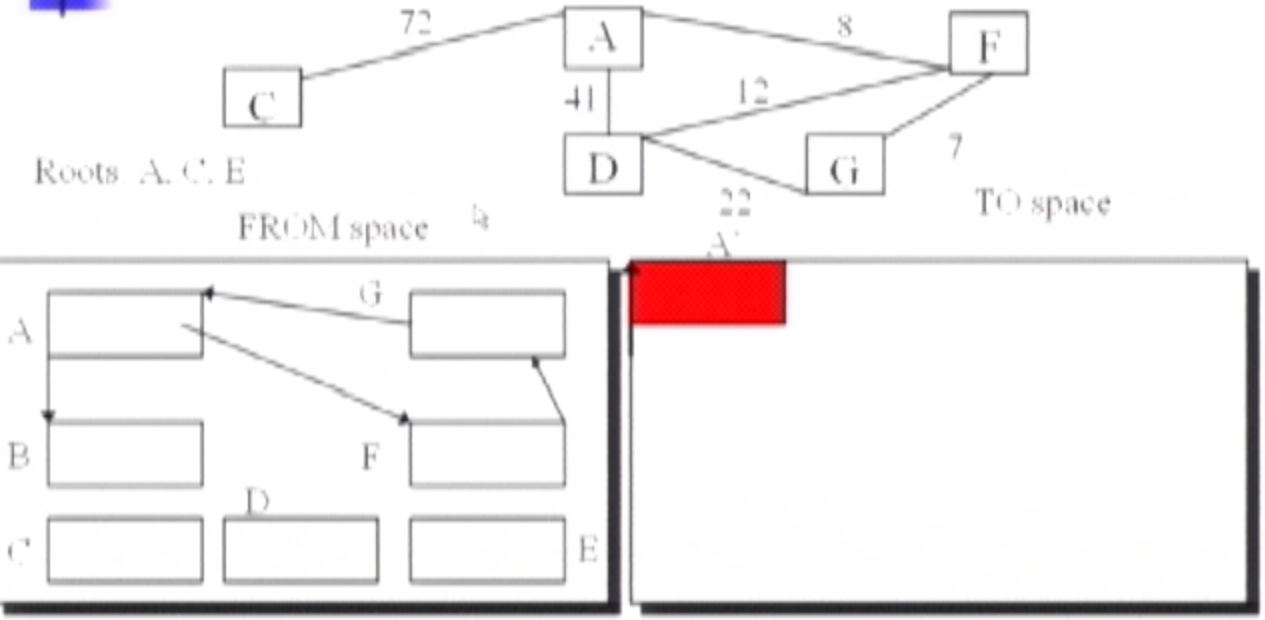

Locality Queue

# Cache-Conscious Copying

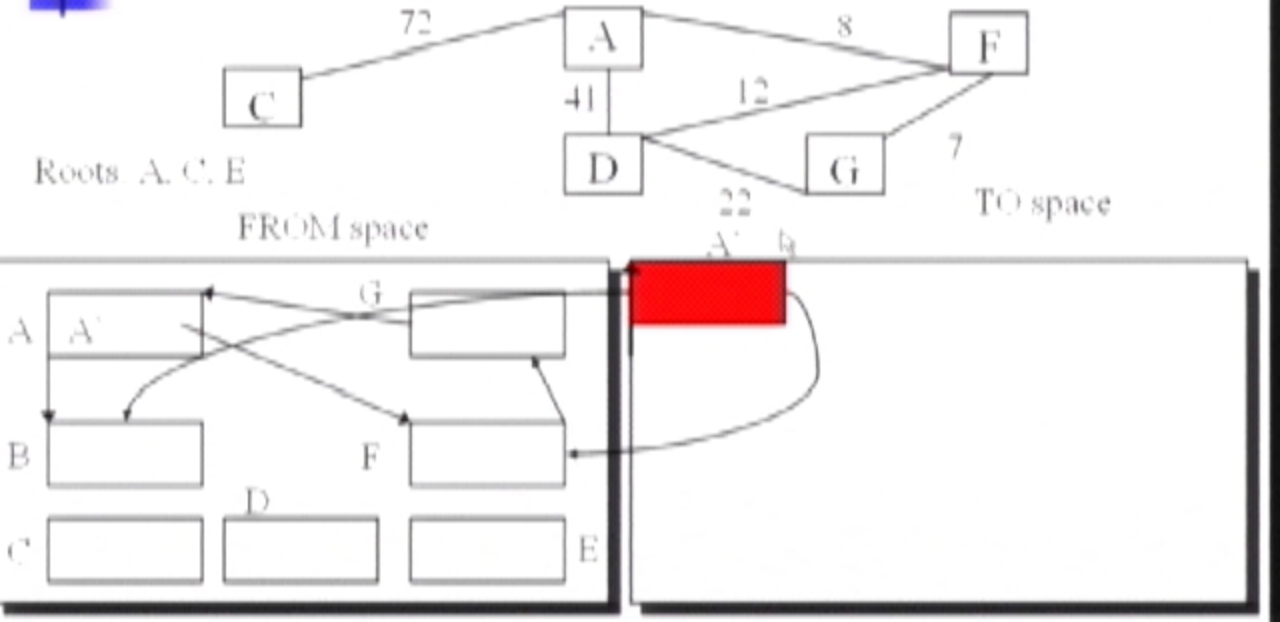


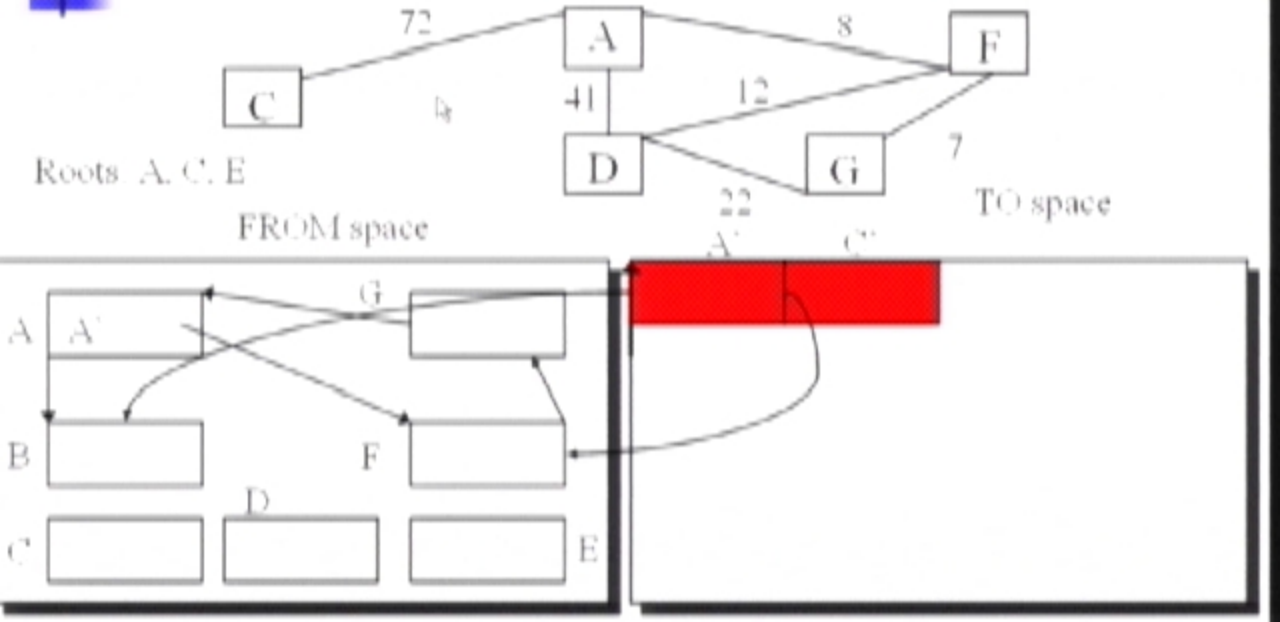
# Cache-Conscious Copying



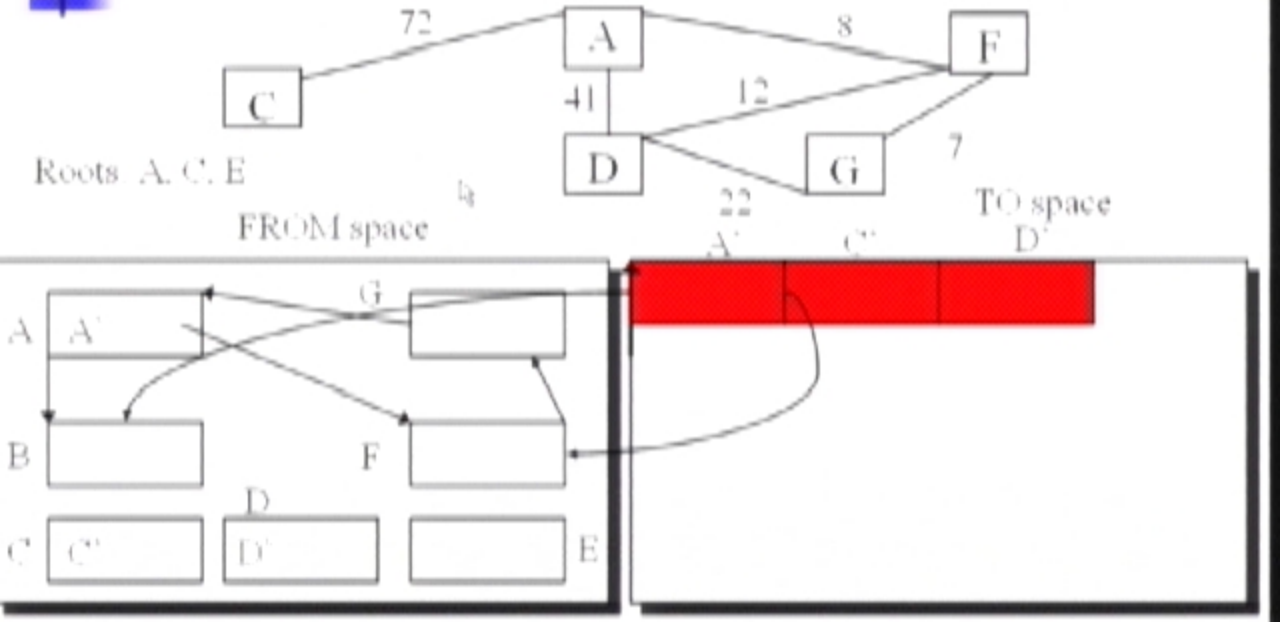
# Cache-Conscious Copying



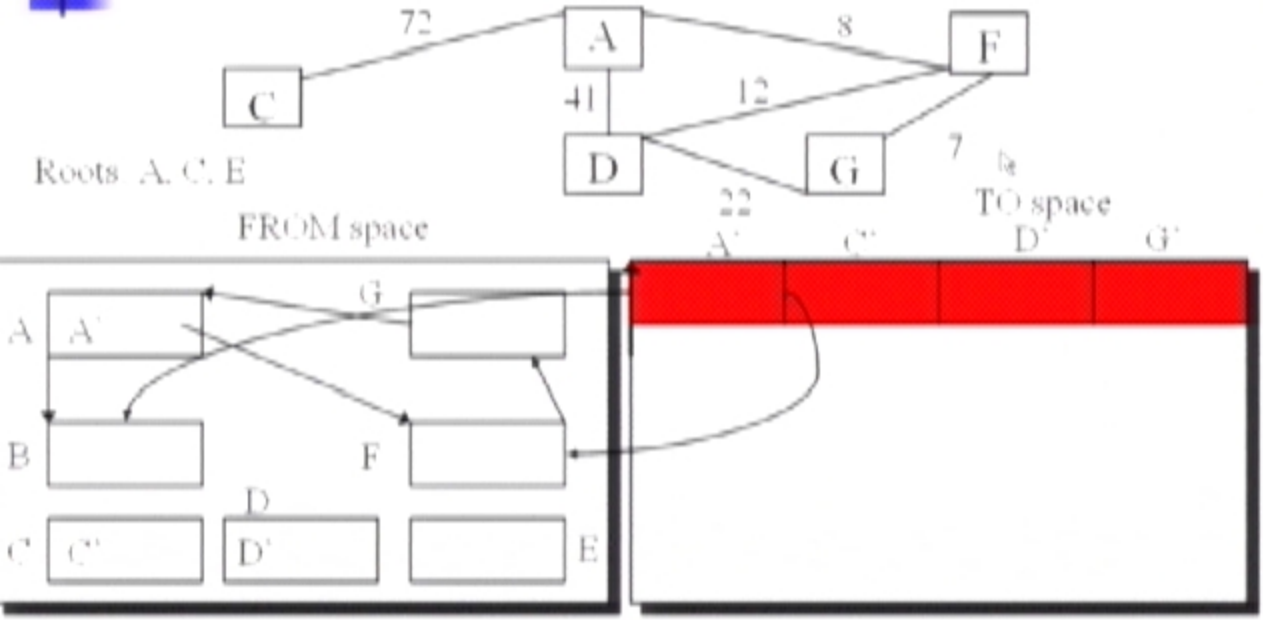
# Cache-Conscious Copying



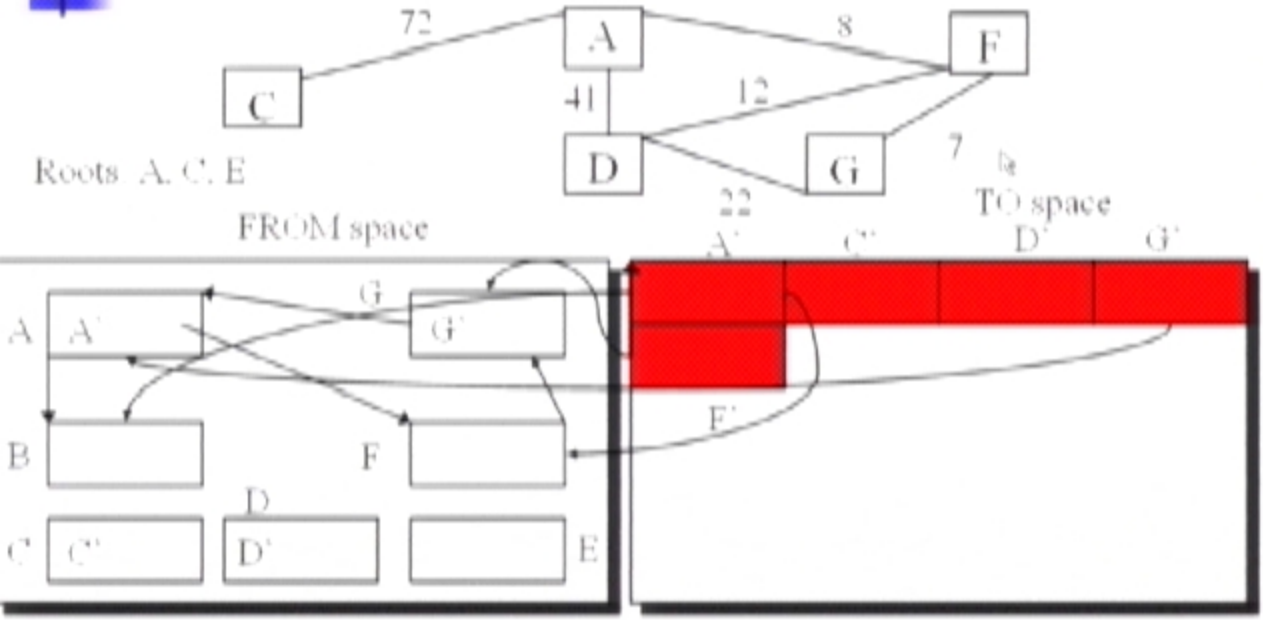
# Cache-Conscious Copying



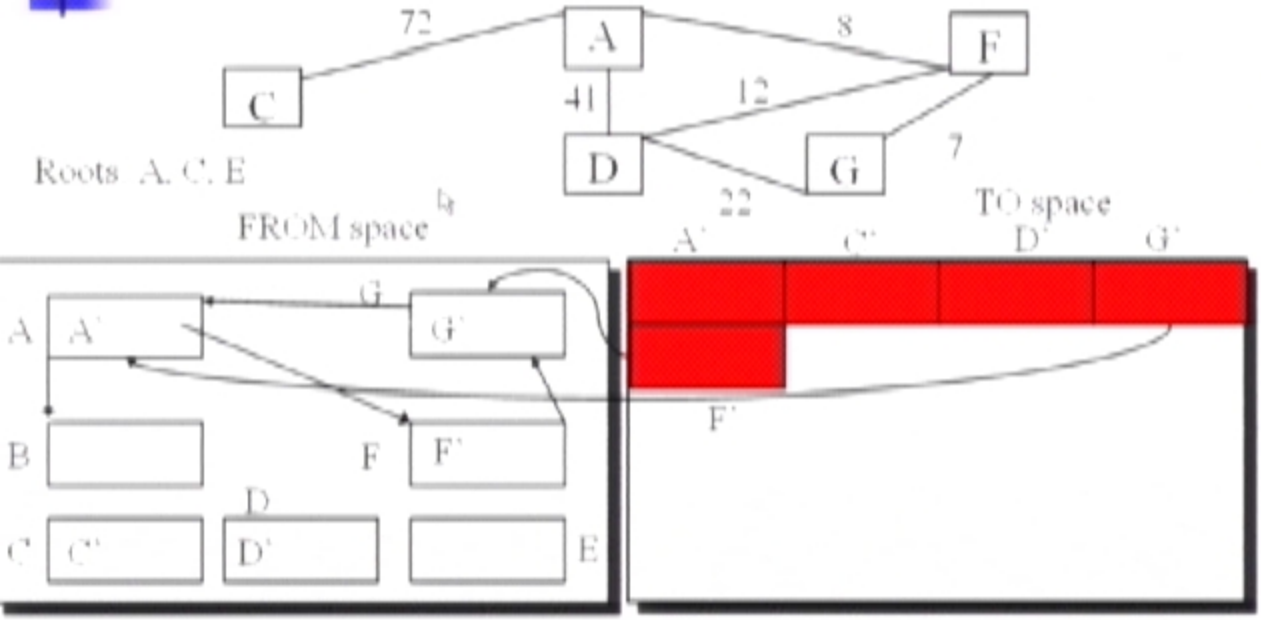
# Cache-Conscious Copying



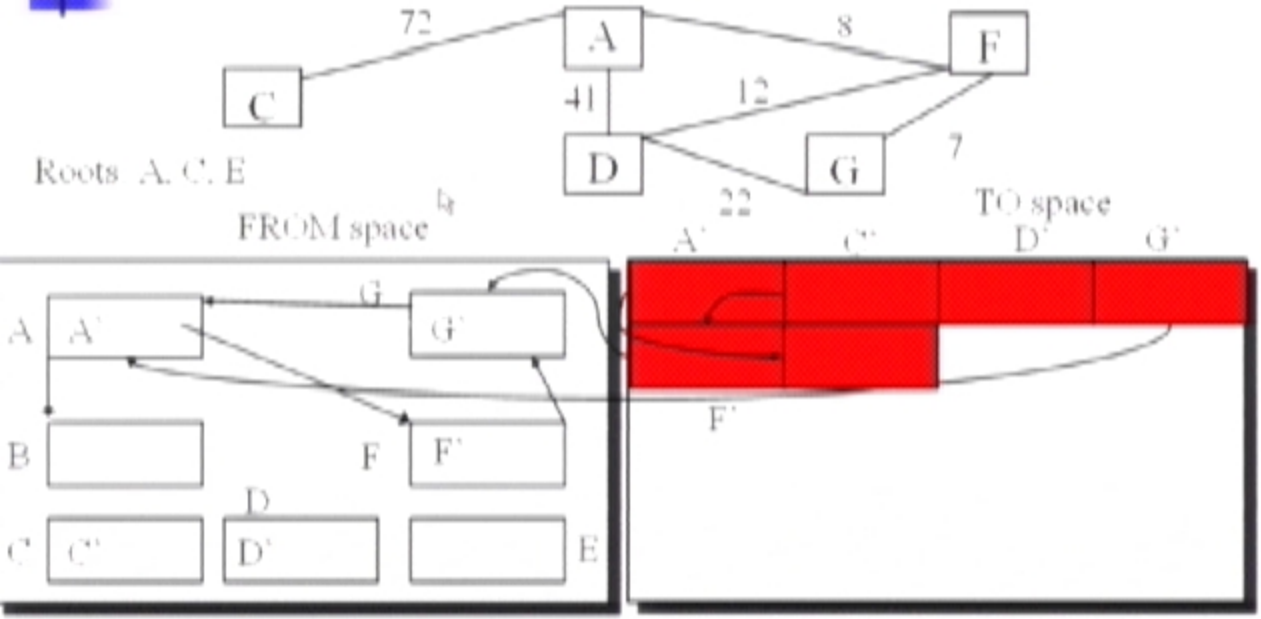
# Cache-Conscious Copying



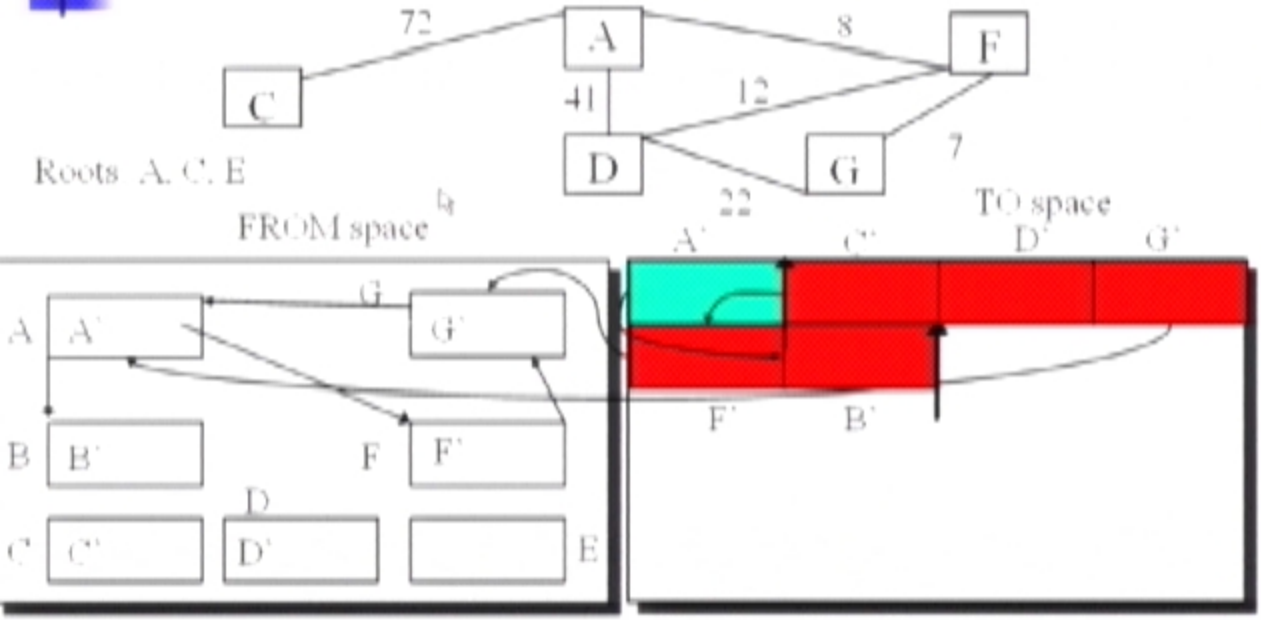
# Cache-Conscious Copying



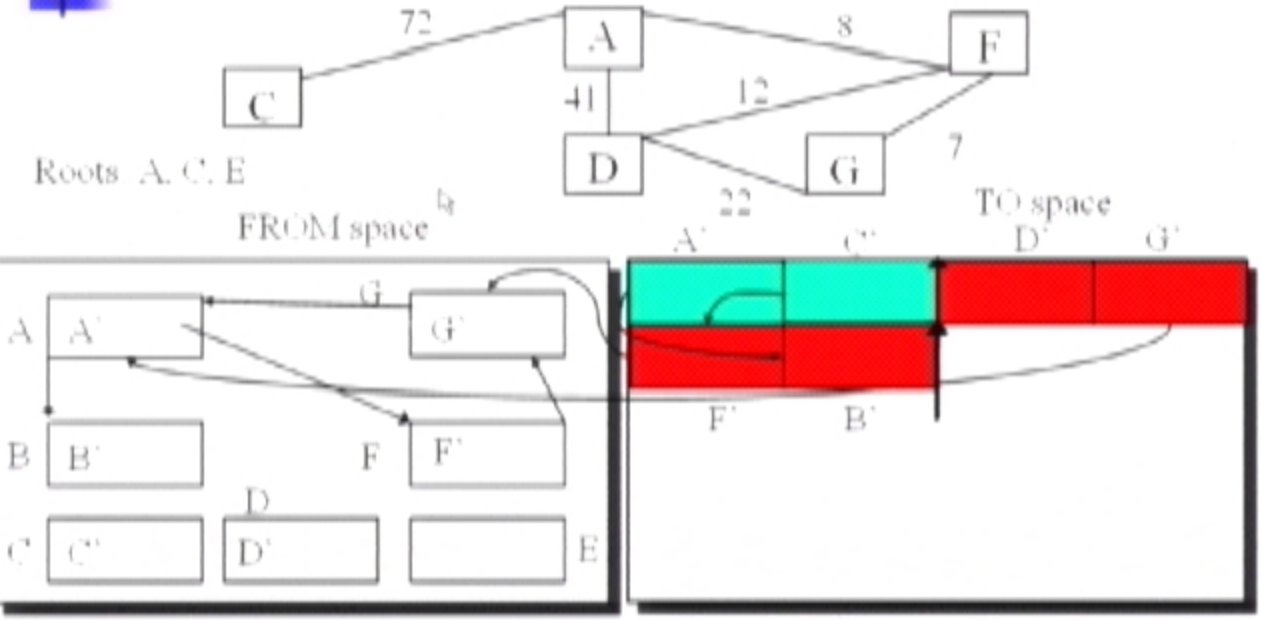
# Cache-Conscious Copying



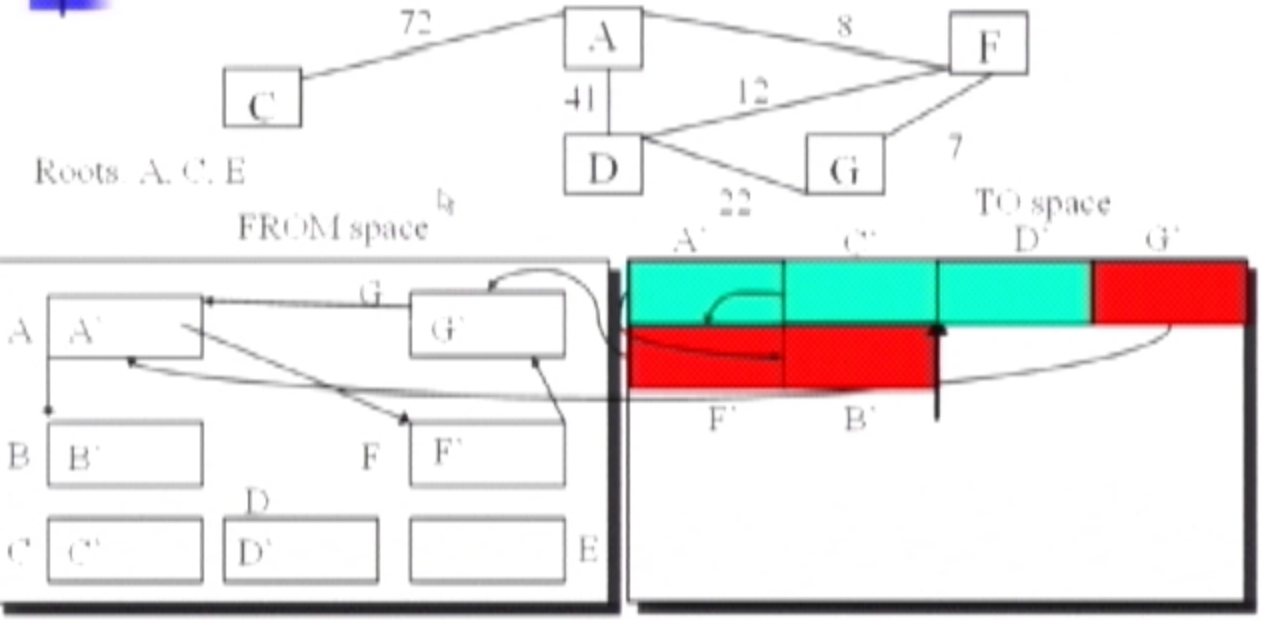
# Cache-Conscious Copying



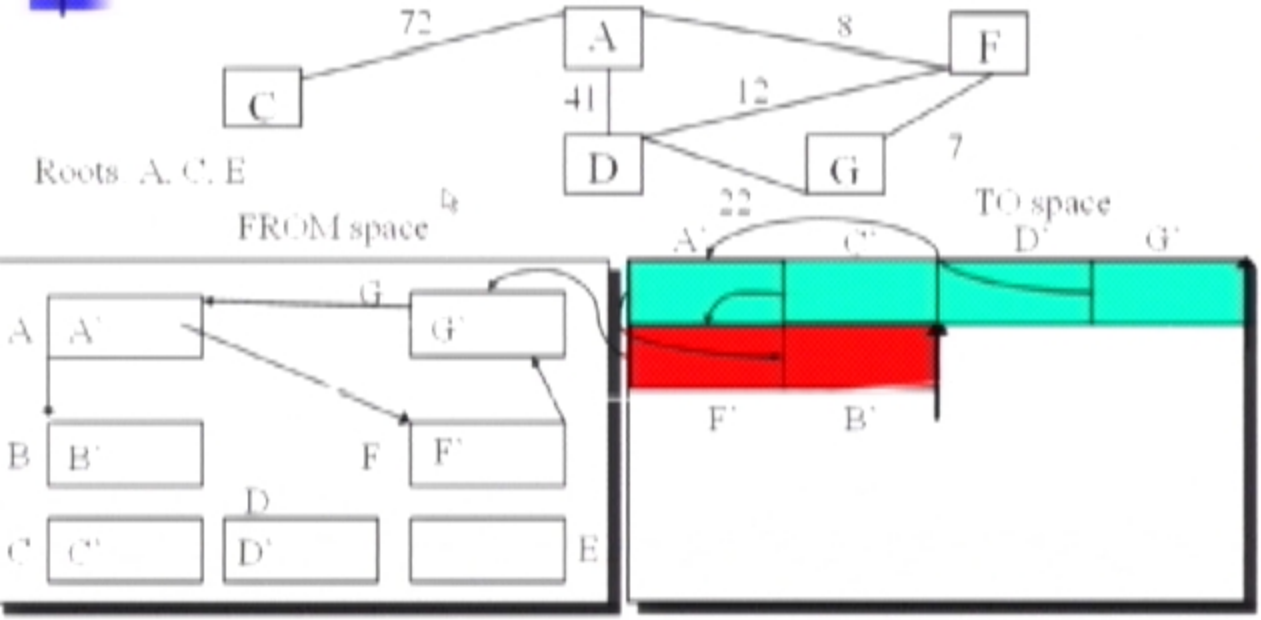
# Cache-Conscious Copying



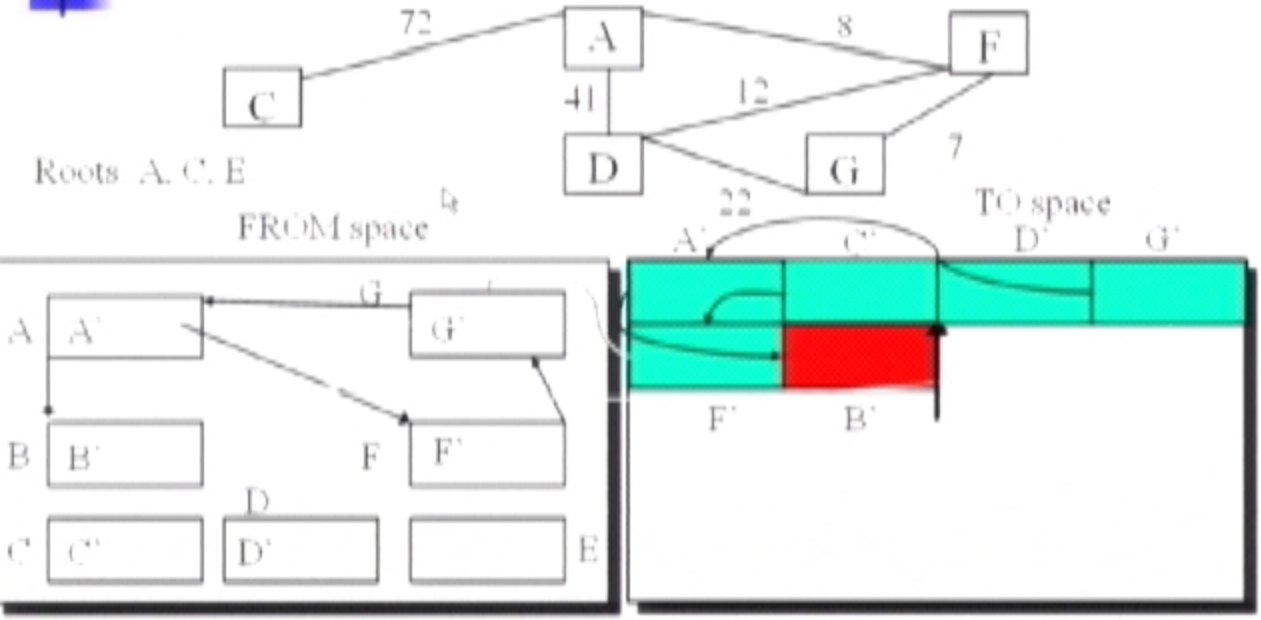
# Cache-Conscious Copying



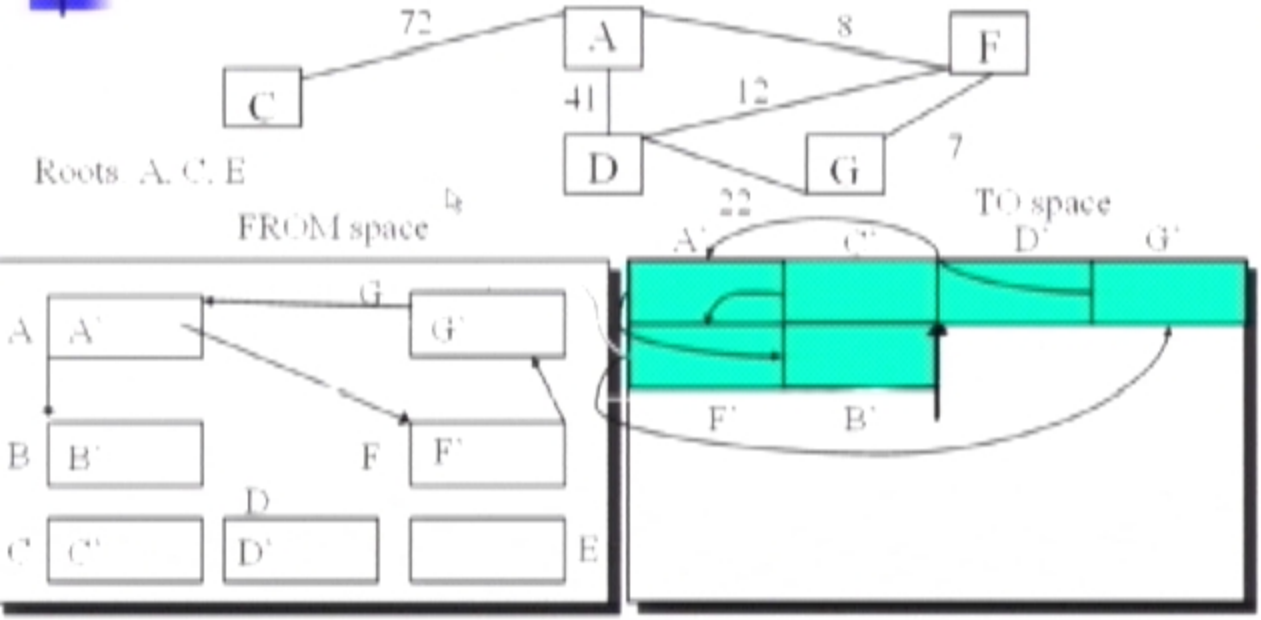
# Cache-Conscious Copying



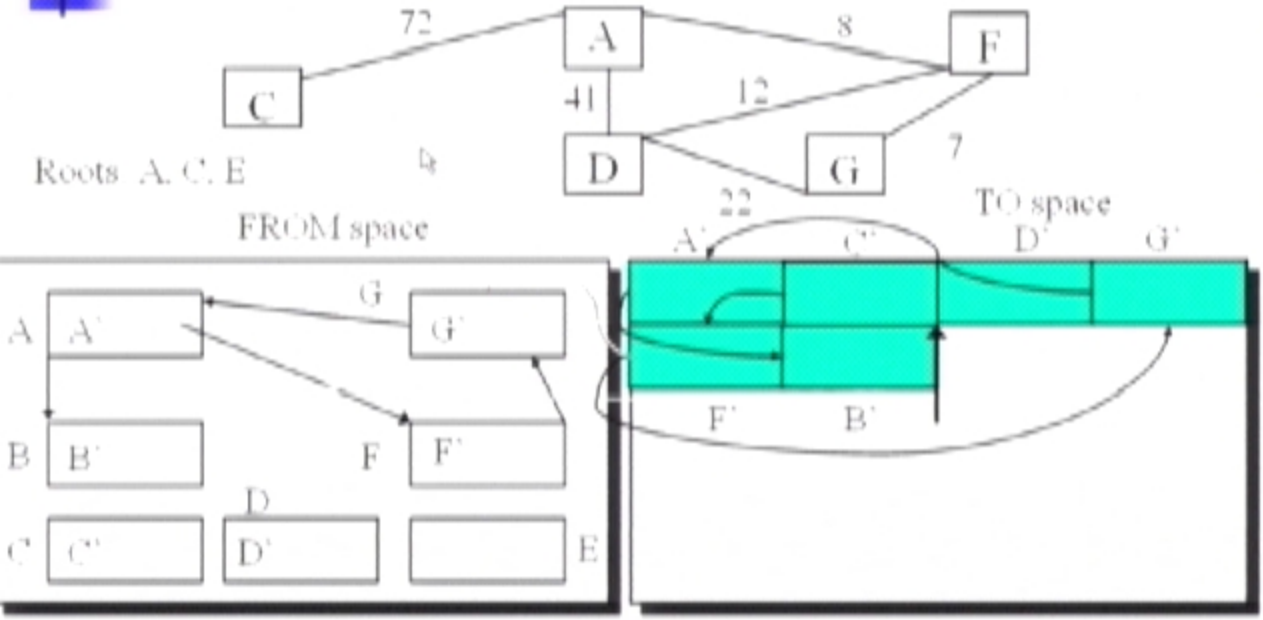
# Cache-Conscious Copying



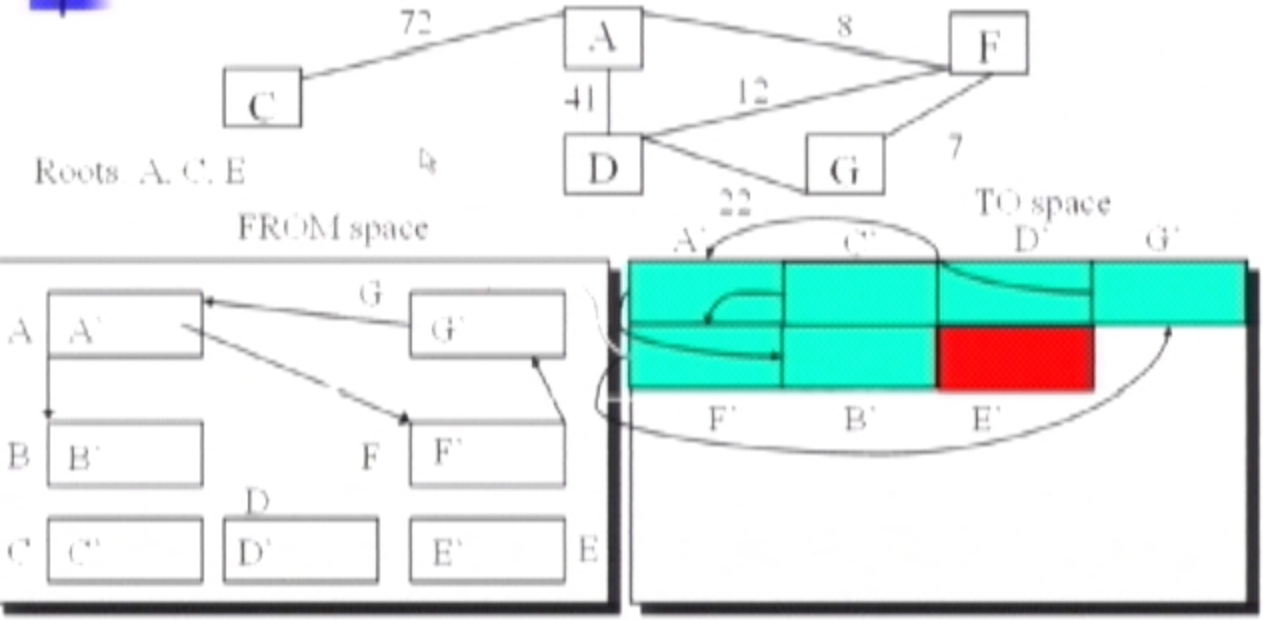
# Cache-Conscious Copying



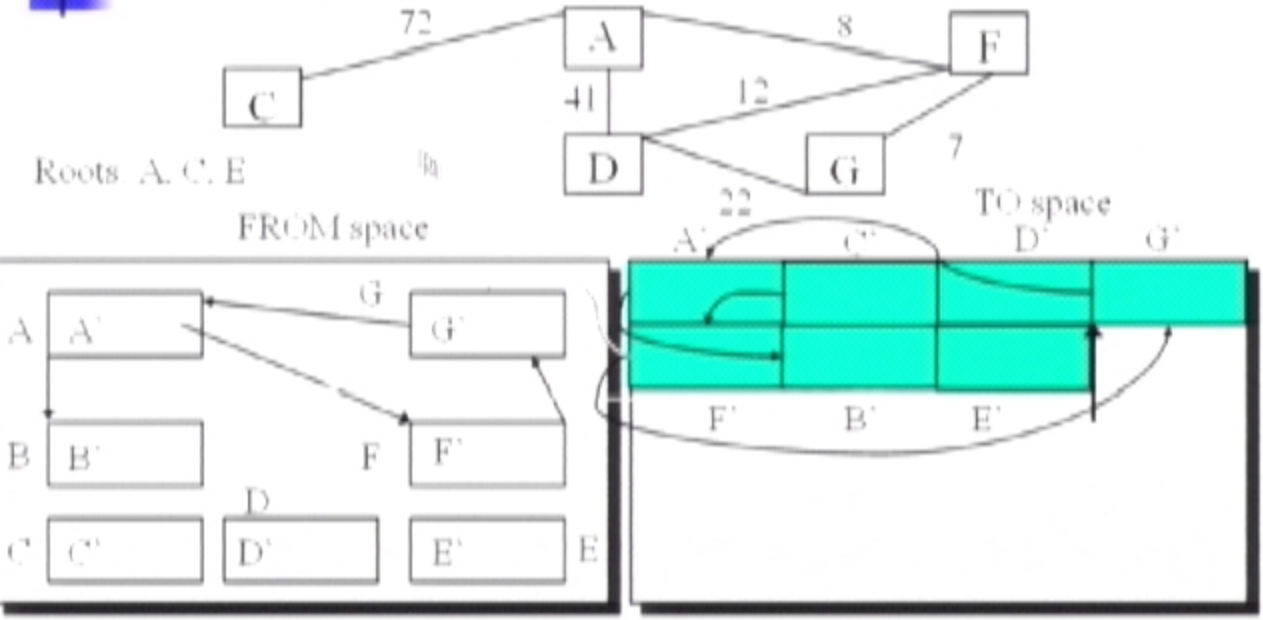
# Cache-Conscious Copying



# Cache-Conscious Copying

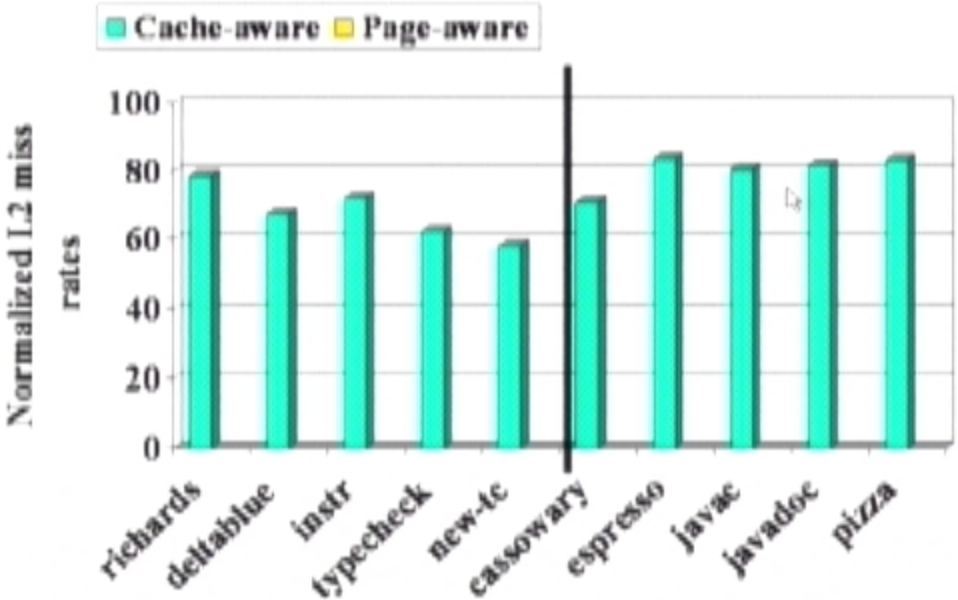


# Cache-Conscious Copying

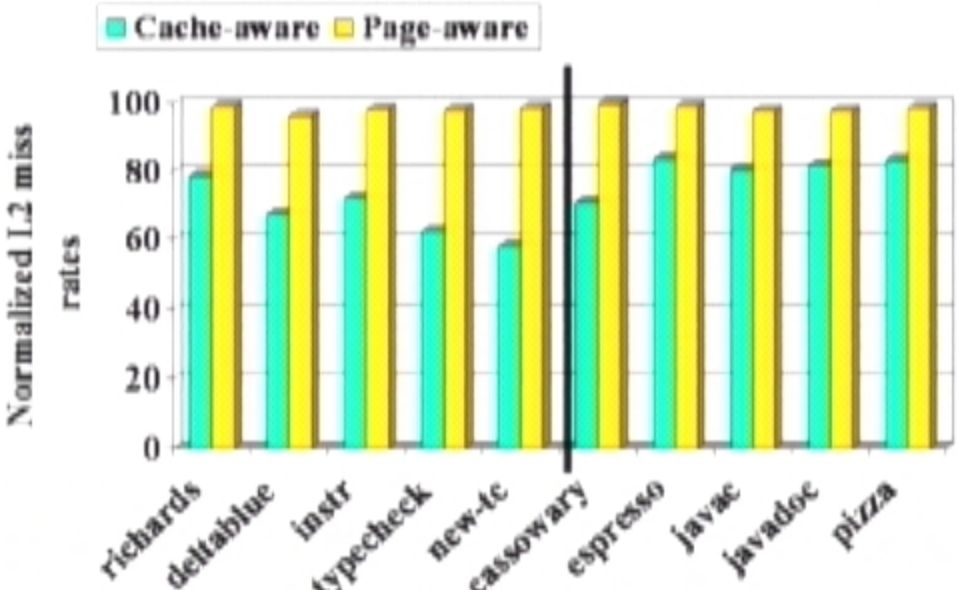


# Cache-Conscious Reorganization

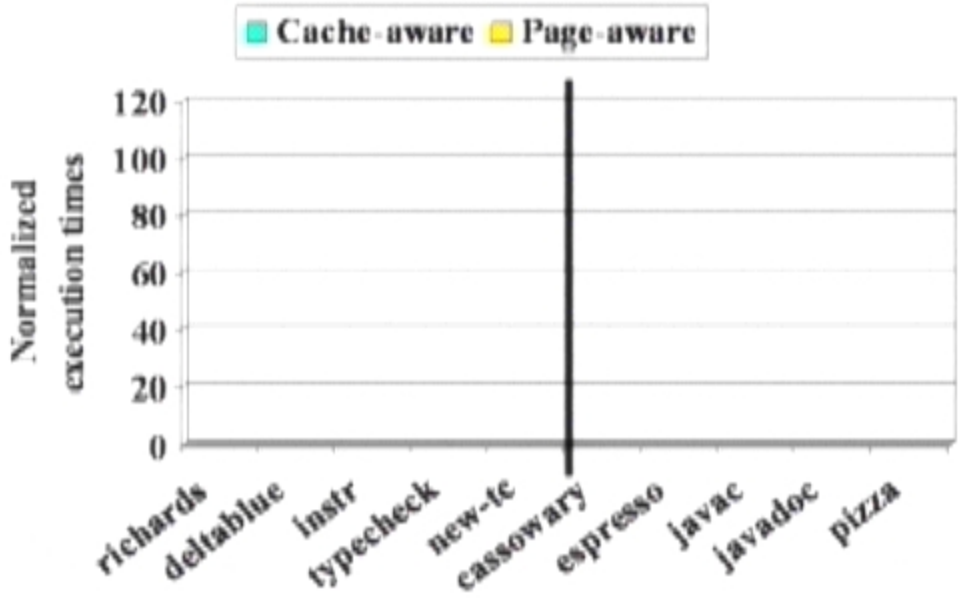
# Cache-Conscious Reorganization



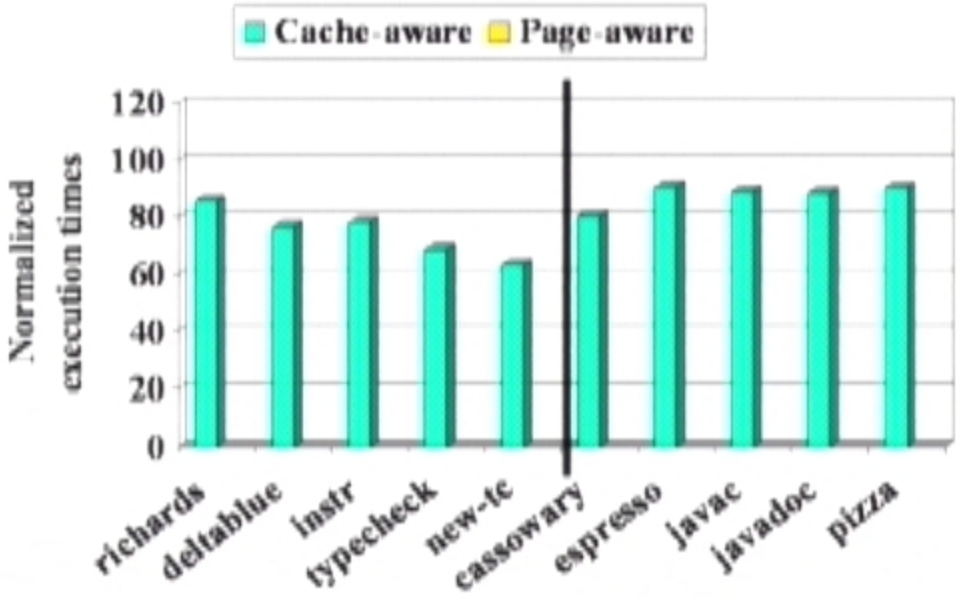
# Cache-Conscious Reorganization



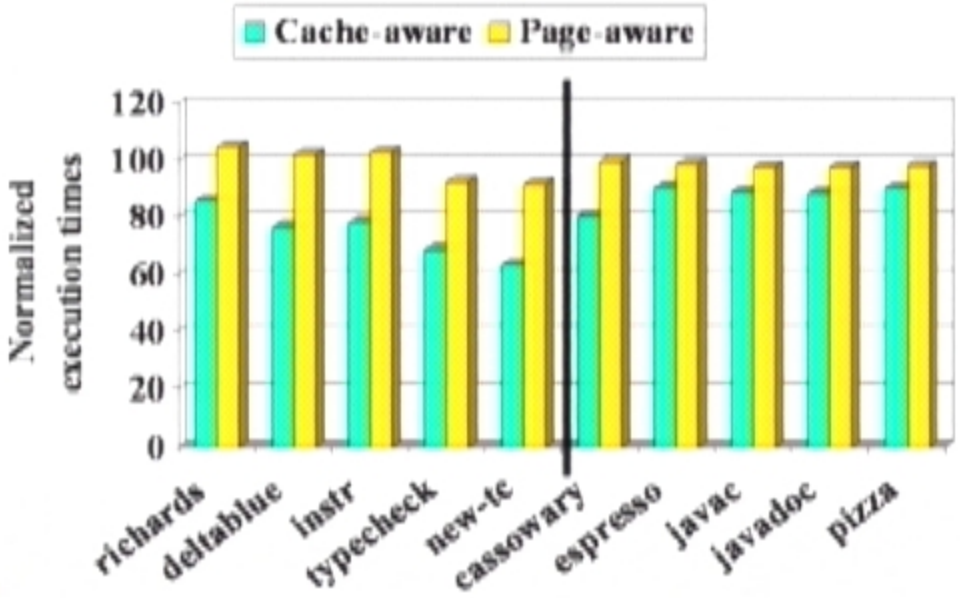
# Cache-Conscious Reorganization



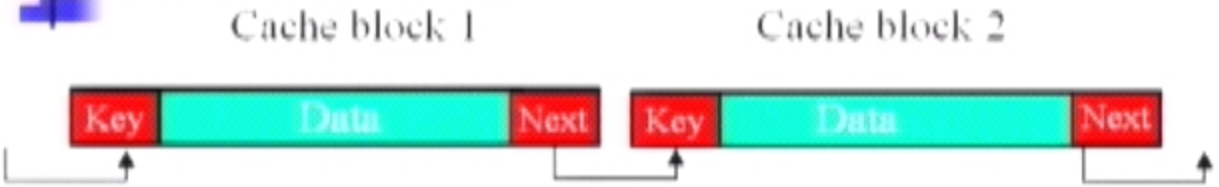
# Cache-Conscious Reorganization



# Cache-Conscious Reorganization

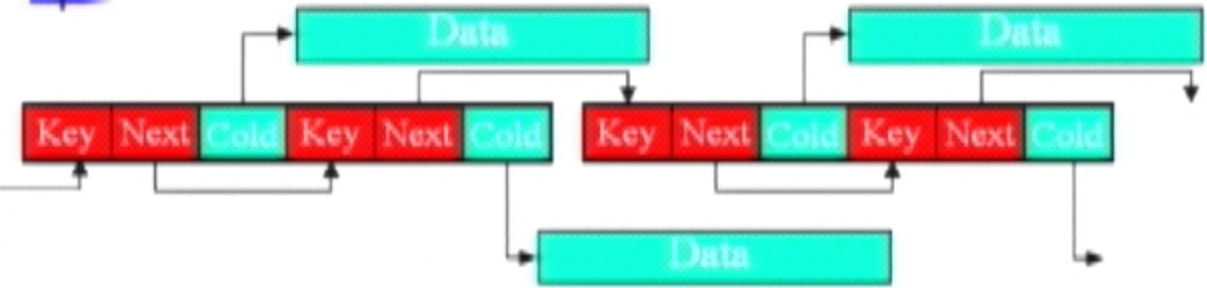


# Hot/Cold Structure Splitting




```
for(p = head; p != NULL; p = p->next){
      if(p->key == key){
            /* Examine p->data */
      }
}
```

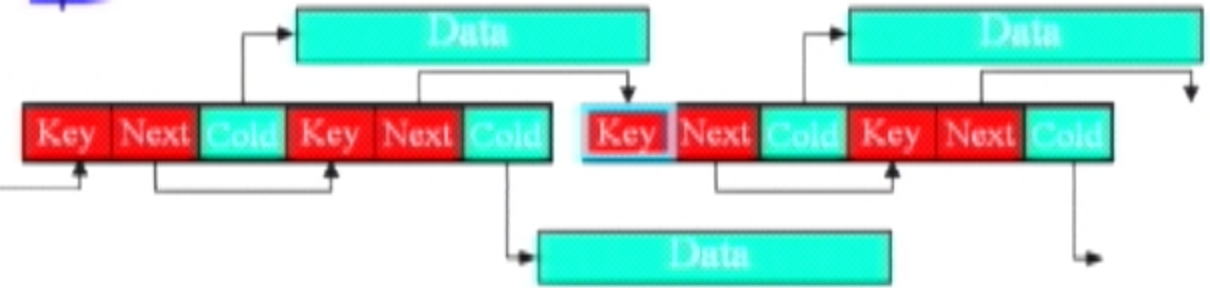
# Hot/Cold Structure Splitting




```
struct list {
 int key;
 struct list *next;
 struct cold_list *cold;
};
```

```
struct cold_list {
 int data[100];
};
```

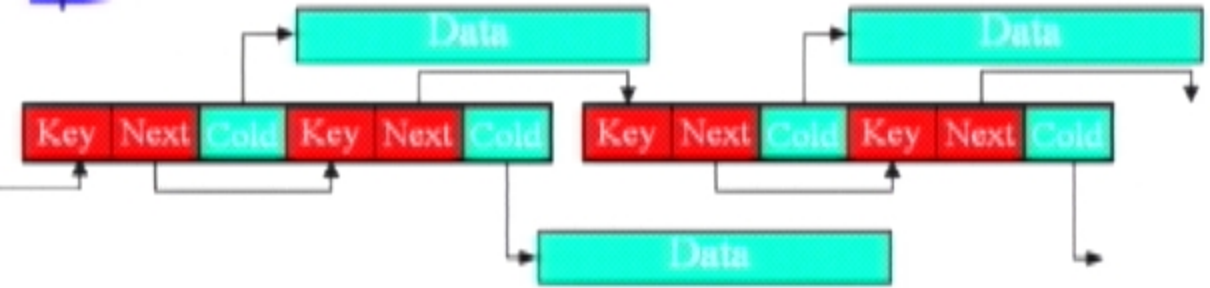
# Hot/Cold Structure Splitting




```
struct list {
 int key;
 struct list *next;
 struct cold_list *cold;
};
```

```
struct cold_list {
 int data[100];
};
```

# Hot/Cold Structure Splitting



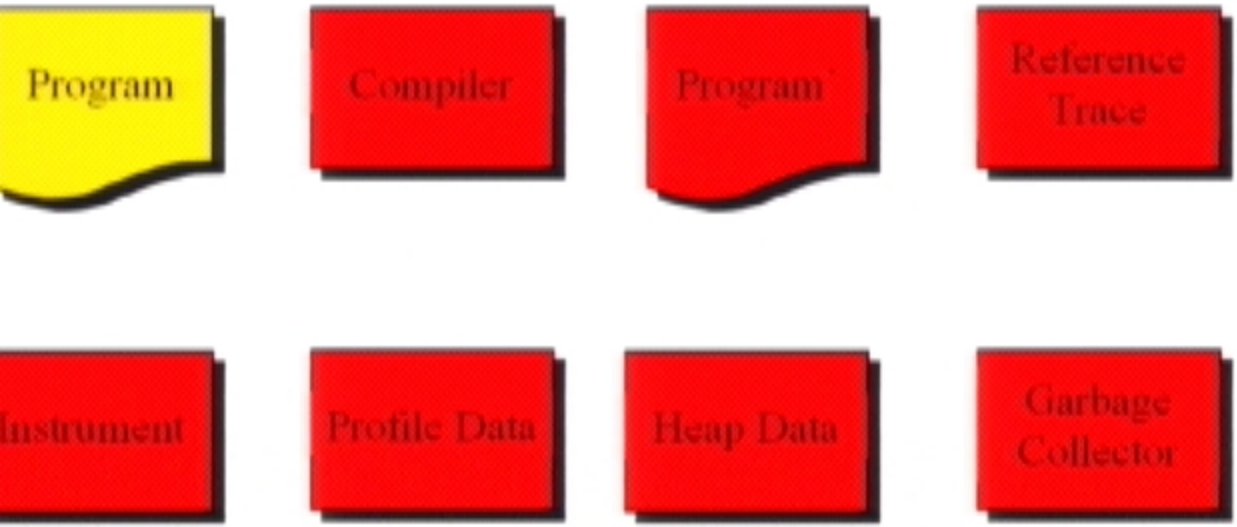

```
struct list {
 int key;
 struct list *next;
 struct cold_list *cold;
};
```

```
struct cold_list {
 int data[100];
};
```

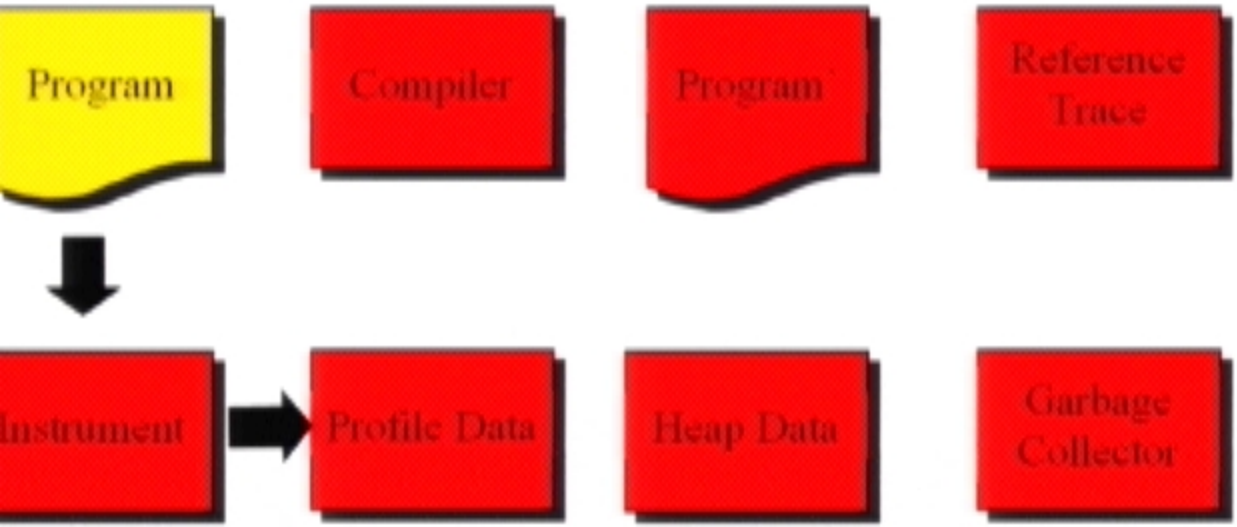
# Hot/Cold Structure Splitting Algorithm

- Profile field access counts
- Identify hot structure fields
- Remove cold fields & place in new structure (link from old)
- Insert additional allocation call for new structure
- Access cold fields through new structure

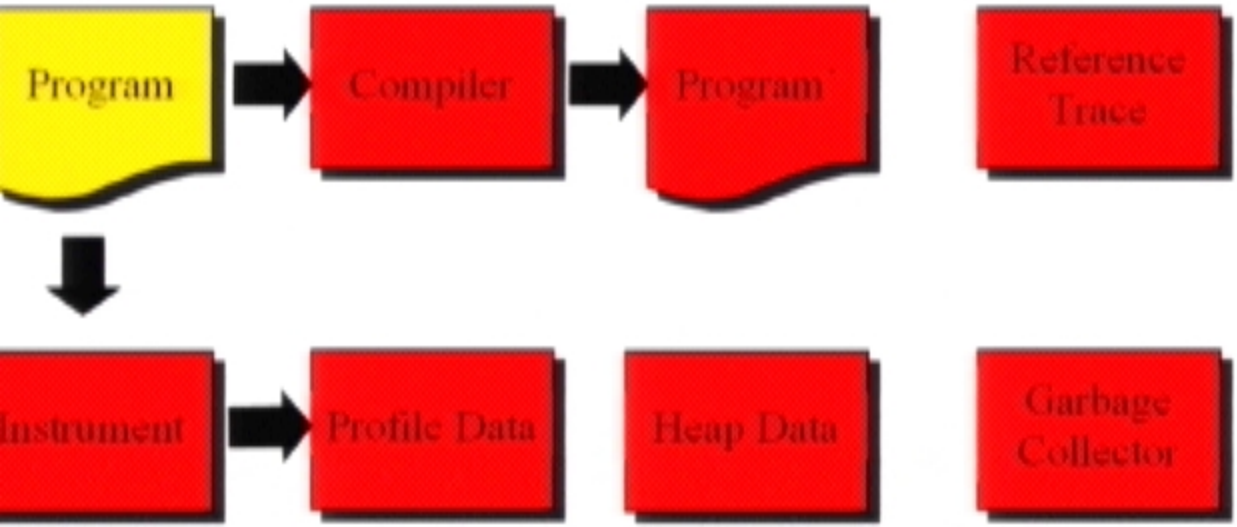
# Cache-Conscious Reorganization (with Structure Splitting)



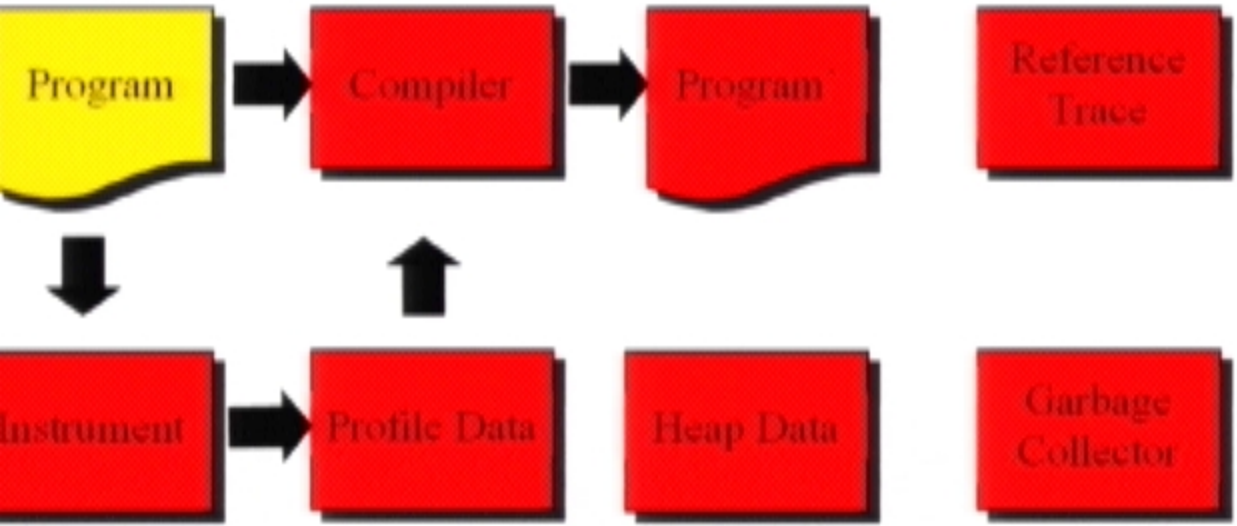
# Cache-Conscious Reorganization (with Structure Splitting)



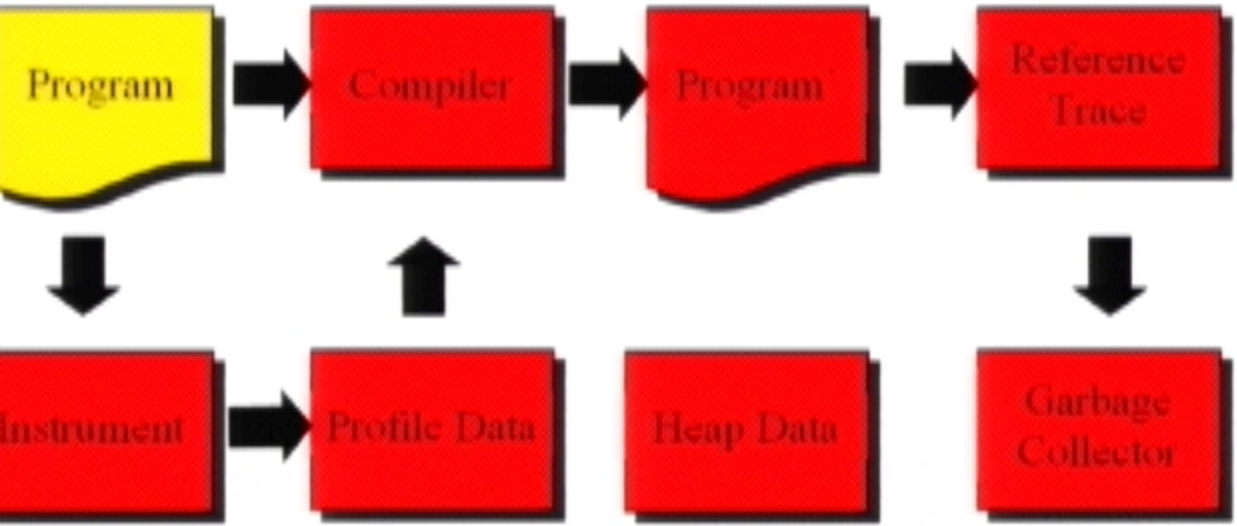
# Cache-Conscious Reorganization (with Structure Splitting)



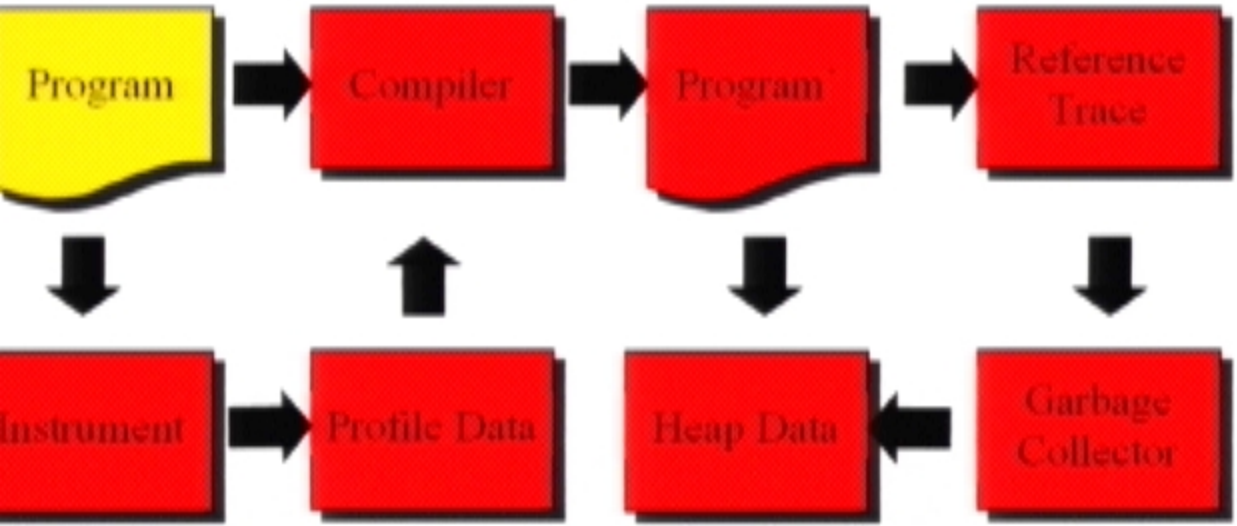
# Cache-Conscious Reorganization (with Structure Splitting)



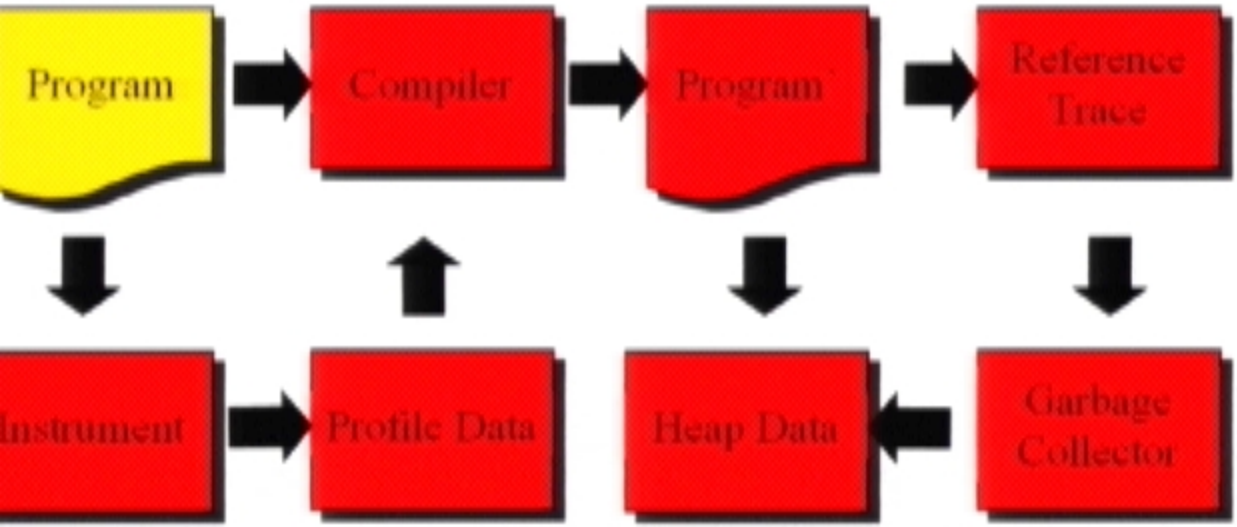
# Cache-Conscious Reorganization (with Structure Splitting)



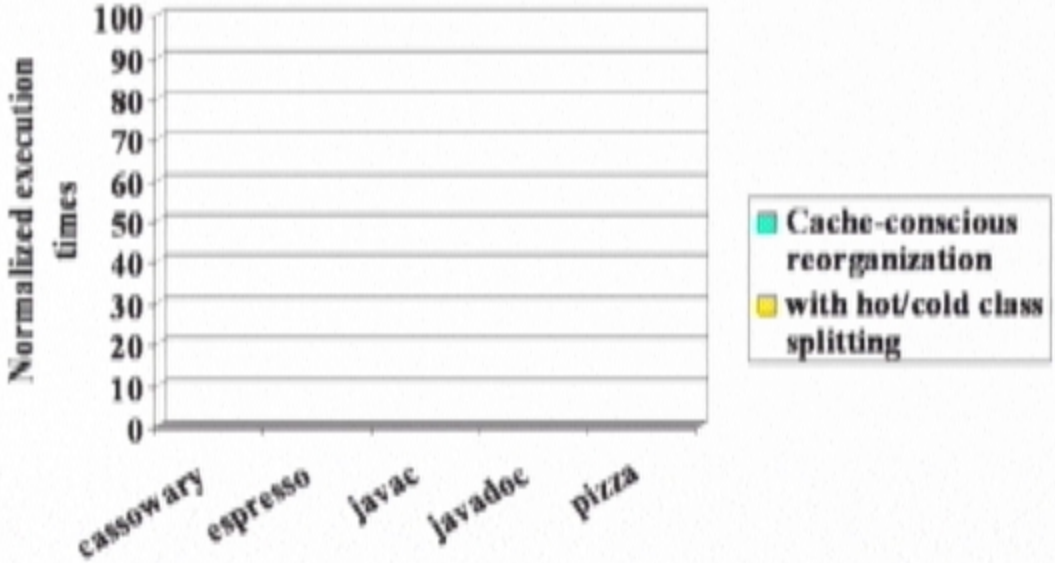
# Cache-Conscious Reorganization (with Structure Splitting)



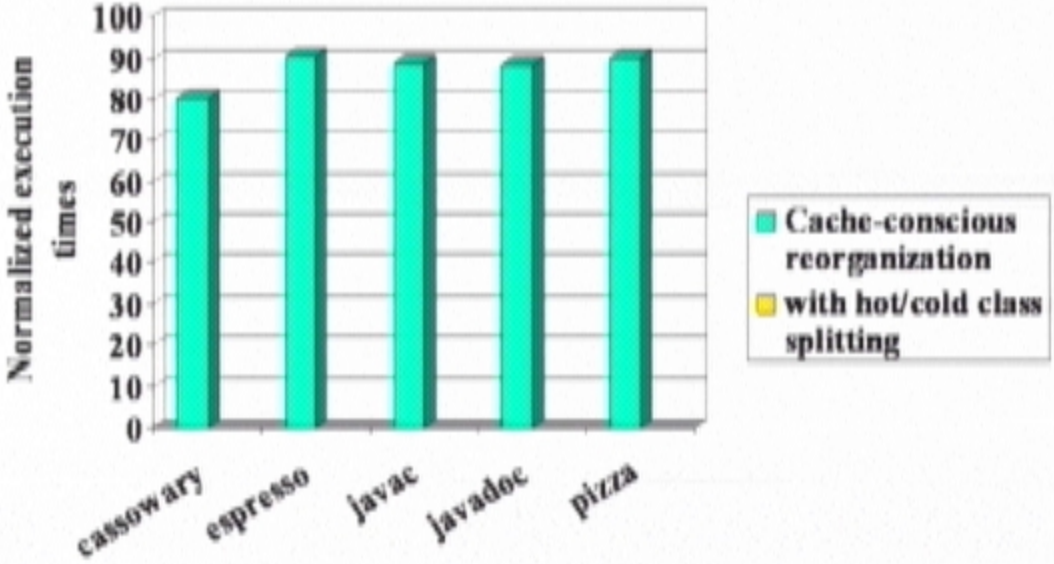
# Cache-Conscious Reorganization (with Structure Splitting)



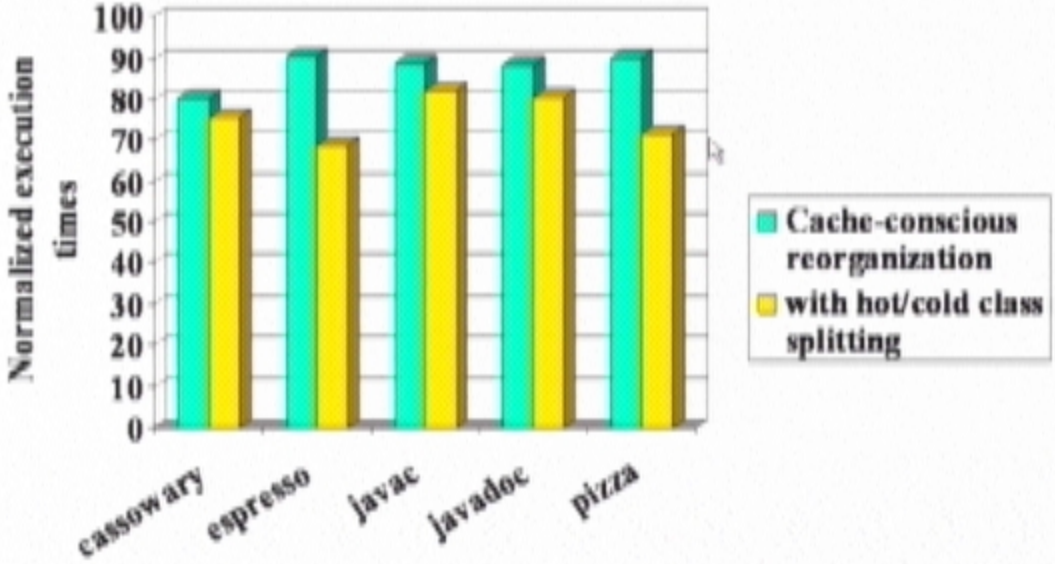
# Hot/Cold Class Splitting



# Hot/Cold Class Splitting



# Hot/Cold Class Splitting



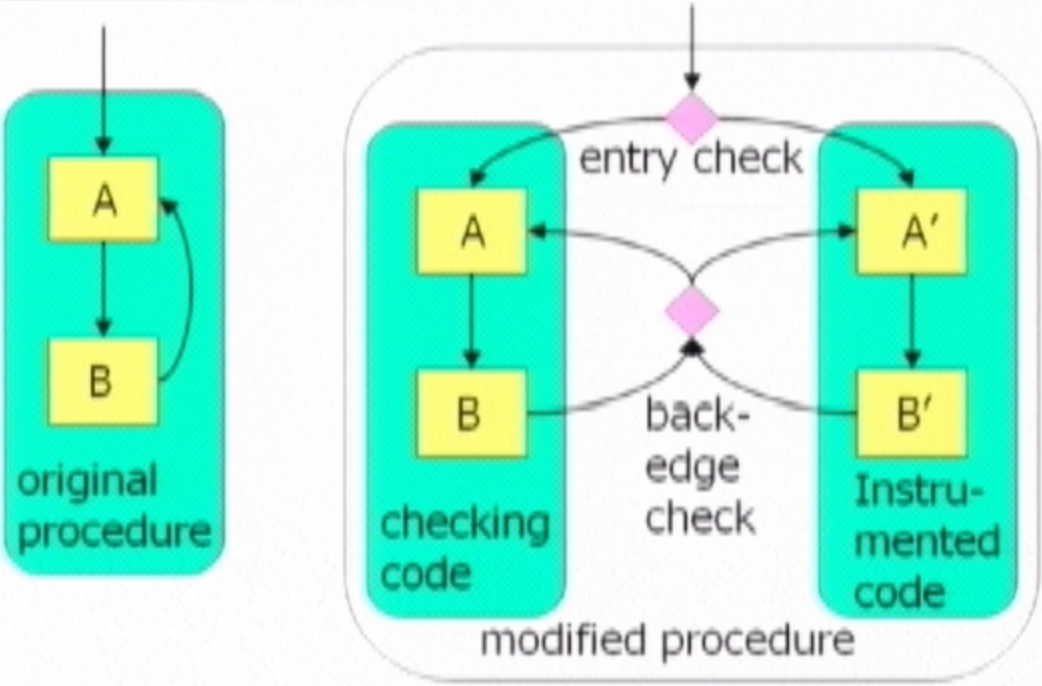
# Limitations

- Low-overhead data reference profiling
  - Requires reserving register
  - Potentially inaccurate compiler analysis
- Object Affinity Graph
  - Approximates temporal relationships
  - Higher accuracy increases processing cost

# Talk Outline

- Background and Motivation
- GC for implementing cache-conscious data layouts
- Low-overhead data reference profiling
- Fast and accurate data layout determination

# Low-overhead data reference profiling



# Low-overhead data reference profiling

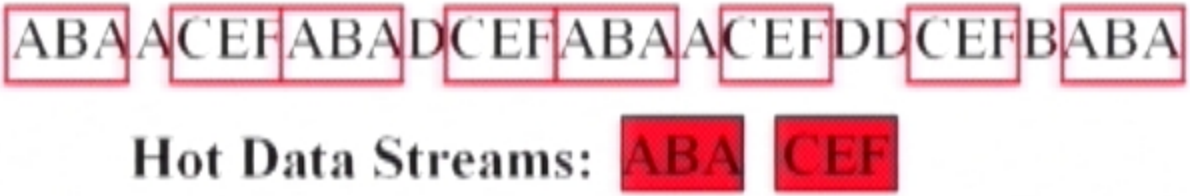
- 0.1% sampling rate provides high accuracy
- 5% overhead at this sample rate
- Base case is highly optimized x86 code

# Talk Outline

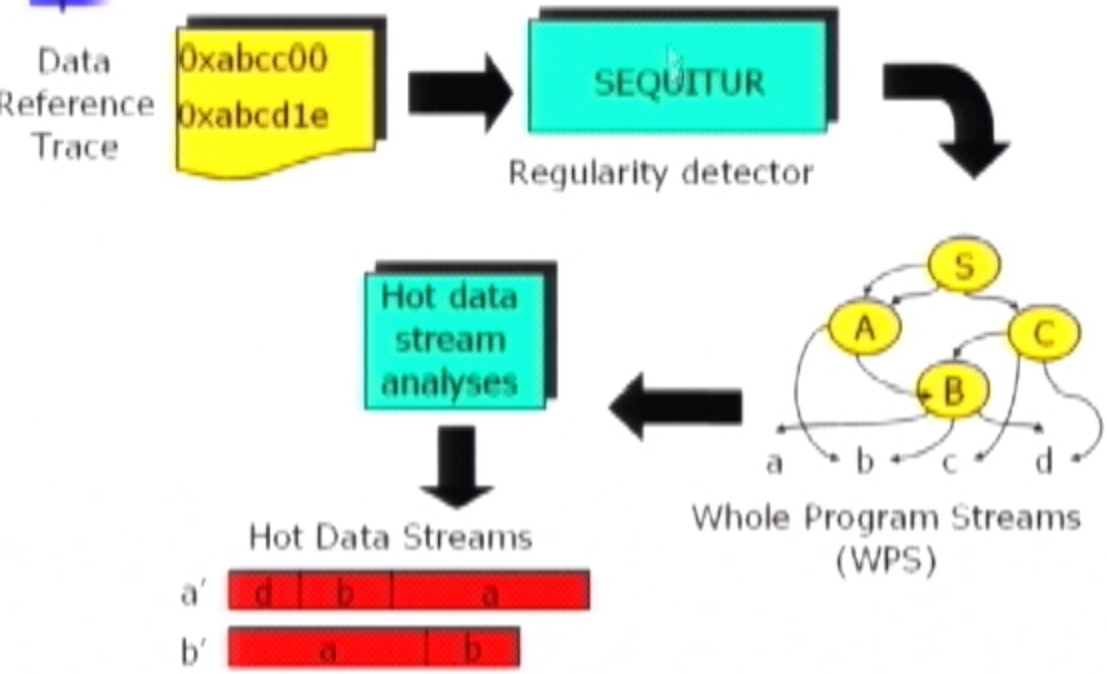
- Background and Motivation
- GC for implementing cache-conscious data layouts
- Low-overhead data reference profiling
- Fast and accurate data layout determination

# Exploitable Locality

- Hot Data Streams:
  Data object sequences that frequently repeat
- Hot=>reference skew + Repetition=>regularity
- Analogous to hot program paths



# Computing Exploitable Locality



# Fast, accurate data layout determination

- Fast linear time algorithms
  - Construct Whole Program Stream representation
  - Detect hot data streams
- Few seconds, yet precise information

# Conclusions

- Cache-conscious layouts offer large benefits
- GC for cache-conscious data layouts
  - Structure layout reorganization
  - Hot/cold splitting
- Practical within context of CLR
  - Low-overhead data reference profiling
  - Fast, accurate data layout determination
- Opportunity for C# to approach performance of C/C++

# Future

- Implement and evaluate in the context of the CLR

# References (http://research.microsoft.com/~trishulc)

- GC for cache-conscious data layout
  - Using GC to implement cache-conscious data layout, ISMM 98
  - Cache-conscious structure definition, PLDI 1999
- Low-overhead data reference profiling
  - Bursty tracing: A low-overhead framework for temporal profiling, FDDO 2001 submission
- Fast, accurate data layout determination
  - Efficient data reference locality abstractions and representations, PLDI 2001